

由Elasticsearch内核开发工程师编写，从源码和设计角度分析Elasticsearch的内部原理，为合理、高效地使用Elasticsearch提供理论指导，并为大规模应用和维护过程中的常见问题提供具体的优化措施和故障诊断方法



Elasticsearch

源码解析与优化实战

张超 / 著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn



版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



作者简介



张 超

长期从事服务端和基础架构等研发工作，对搜索、分布式系统、高性能网络服务有浓厚的兴趣，喜欢探究技术本质，喜欢分析有深度的问题。目前就职于360企业安全集团基础大数据团队，负责平台内核研发工作。





Elasticsearch

源码解析与优化实战

张超 / 著

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING





内 容 简 介

本书介绍了 Elasticsearch 的系统原理,旨在帮助读者了解其内部原理、设计思想,以及在生产环境中如何正确地部署、优化系统。系统原理分两方面介绍,一方面详细介绍主要流程,例如启动流程、选主流程、恢复流程;另一方面介绍各重要模块的实现,以及模块之间的关系,例如 gateway 模块、allocation 模块等。本书的最后一部分介绍如何优化写入速度、搜索速度等大家关心的实际问题,并提供了一些诊断问题的方法和工具供读者参考。

本书适合对 Elasticsearch 进行改进的研发人员、平台运维人员,对分布式搜索感兴趣的朋友,以及在使用 Elasticsearch 过程中遇到问题的人们。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

Elasticsearch 源码解析与优化实战 / 张超著. —北京:电子工业出版社,2018.11
ISBN 978-7-121-35216-4

I. ①E… II. ①张… III. ①搜索引擎—程序设计 IV. ①TP391.3

中国版本图书馆 CIP 数据核字(2018)第 238923 号

责任编辑:陈晓猛

印 刷:三河市良远印务有限公司

装 订:三河市良远印务有限公司

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱

邮编:100036

开 本:787×980 1/16 印张:22.5

字数:432 千字

版 次:2018 年 11 月第 1 版

印 次:2018 年 11 月第 1 次印刷

定 价:89.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:010-51260888-819, faq@phei.com.cn。





前言

我们可以在不关心原理的情况下使用 Elasticsearch（以下简称 ES），但要想用好 ES，就必须熟知其内部原理。

为什么要阅读代码？在传统软件行业，技术文档非常丰富。当开展一个项目时，从需求分析，到概要设计、详细设计，每个步骤都有相应的文档，从项目的整体架构、技术方案选型，到流程图、类图，细化到每个接口及参数。在这种情况下，想要搞清楚系统原理，并不需要阅读代码，文档上什么都有。但是互联网产品迭代快，技术文档不全，想要搞清楚原理，只能阅读代码，相当于从代码中逆向理解设计思想。

通过分析源码，我们可以有以下收获：

理解设计思想 当我们面临要解决的问题或实现的目标时，往往有多种方案可以选择。无论表面上看起来多么简单的架构，其背后都经过了深思熟虑。思考一下为什么使用现在的方案？有没有更好的解决方案？

探究内部机制的原理 某个技术点是怎么实现的？

搞明白执行流程 某个过程是什么样的，都做了什么？有几步？先做什么，后做什么？

熟悉代码结构 如果需要二次开发，则给出代码入口和调用关系，有时候找到某个逻辑的代码实现要花很多时间。

学以致用 借鉴其设计理念，掌握其解决问题的方式和方法，将来面对类似的问题时可以参考。

本书结构

本书由四部分组成，第一部分为基础知识和环境准备（第 1~2 章）；第二部分介绍 ES 的主要流程（第 3~10 章），包括集群启动流程、节点启动/关闭流程、选主流程、读写流程、搜索流程和索引恢复流程；第三部分主要介绍重要内部模块（第 11~17 章），包括 gateway 模块、allocation 模块、Snapshot 模块、Cluster 模块、Transport 模块和 ThreadPool 模块等；第四部分介





绍优化和诊断方法（第 18~22 章），包括写入速度优化、搜索速度优化、磁盘使用量优化，以及在生产环境中的实际应用建议，第 22 章介绍常用的问题诊断方法，排查集群遇到的问题。

术语约定

ES 中有一些特有的概念，这些概念对应的中文翻译约定如下：

- 分片（shard）；
- 主分片（primary shard），简称 P；
- 分片副本（特指数据的一个分片，无论主分片，还是副分片）；
- 副分片（replica shard），简称 R；
- 分片分配（shard allocation）；
- 集群状态（cluster state）；
- 分配决策（allocation decision）；
- 分配感知（allocation awareness）；
- 分配标识（allocation IDs）；
- 追踪（tracking）；
- 事务日志（translog）；
- 同步集合（in-sync set）。

行文约定

虽然本书是一本源码分析类图书，但原则上尽量少贴代码，引用的代码只是为了说明原理，因此所引用的代码并不保证和源码完全一致，对非核心逻辑有所删减，同时在代码块中，函数参数可能被省略，省略的函数参数用“...”表示，如：

```
executeBulk(...);
```

在引用代码中的某个方法时，使用#号分隔类名与方法名：

类名#方法名

一个索引由许多分片组成。我们用如下方式表示索引 website 的第 0 个分片：

```
website[0]
```





联系

读者有任何意见和建议都可以联系作者，邮箱：elasticsearchbook@163.com。

本书配套网站：www.elasticsearchbook.cn。

致谢

感谢李欣杰和郭东东，他们带我走进搜索领域；感谢韩洪伟，他让我学到了很多搜索系统的知识。欣杰和老韩都是资深的搜索架构师，能够和优秀的团队共事是我的荣幸。感谢 ES 团队的同事段军义，我们互相学习，一起解决了很多麻烦的问题。感谢出版社的策划编辑陈晓猛先生，他为本书的写作提供了很多建设性意见，并且耐心地编校了本书，让本书得以顺利出版。

感谢我的妻子和三岁的女儿，我爱你们！

张超

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/35216>





目录

第 1 章 走进 Elasticsearch.....	1
1.1 基本概念和原理	1
1.1.1 索引结构	2
1.1.2 分片 (shard)	2
1.1.3 动态更新索引	4
1.1.4 近实时搜索	5
1.1.5 段合并	5
1.2 集群内部原理	6
1.2.1 集群节点角色	6
1.2.2 集群健康状态	8
1.2.3 集群状态	8
1.2.4 集群扩容	8
1.3 客户端 API.....	9
1.4 主要内部模块简介	10
1.4.1 模块结构	11
1.4.2 模块管理	12
第 2 章 准备编译和调试环境.....	13
2.1 编译源码	13
2.1.1 准备 JDK 和 Gradle	13
2.1.2 下载源代码	13
2.1.3 编译项目, 打包	14
2.1.4 将工程导入 IntelliJ IDEA.....	14
2.2 调试 Elasticsearch	16
2.2.1 本地运行、调试项目	16
2.2.2 远程调试	18
2.3 代码书签和断点组	19





第 3 章 集群启动流程.....	21
3.1 选举主节点	22
3.2 选举集群元信息	22
3.3 allocation 过程.....	23
3.4 index recovery	24
3.5 集群启动日志	25
3.6 小结	26
第 4 章 节点的启动和关闭	28
4.1 启动流程做了什么	28
4.2 启动流程分析	28
4.2.1 启动脚本	28
4.2.2 解析命令行参数和配置文件	29
4.2.3 加载安全配置	30
4.2.4 检查内部环境	30
4.2.5 检测外部环境	30
4.2.6 启动内部模块	33
4.2.7 启动 keepalive 线程	33
4.3 节点关闭流程	34
4.4 关闭流程分析	34
4.5 分片读写过程中执行关闭.....	36
4.6 主节点被关闭	36
4.7 小结	36
第 5 章 选主流程.....	38
5.1 设计思想	38
5.2 为什么使用主从模式	38
5.3 选举算法	39
5.4 相关配置	39
5.5 流程概述	41
5.6 流程分析	41
5.6.1 选举临时 Master	42
5.6.2 投票与得票的实现	46
5.6.3 确立 Master 或加入集群	46
5.7 节点失效检测	47
5.7.1 NodesFaultDetection 事件处理.....	47





5.7.2 MasterFaultDetection 事件处理.....	48
5.8 小结	49
第 6 章 数据模型	50
6.1 PacificA 算法	50
6.1.1 数据副本策略	51
6.1.2 配置管理	52
6.1.3 错误检测	52
6.2 ES 的数据副本模型	53
6.2.1 基本写入模型	53
6.2.2 写故障处理	54
6.2.3 基本读取模型	54
6.2.4 读故障处理	55
6.2.5 引申的含义	55
6.2.6 系统异常	56
6.3 Allocation IDs.....	56
6.3.1 安全地分配主分片	56
6.3.2 将分配标记为陈旧	57
6.3.3 一个例子	58
6.3.4 不会丢失全部	63
6.4 Sequence IDs	64
6.4.1 Primary Terms 和 Sequence Numbers	64
6.4.2 本地及全局检查点	66
6.4.3 用于快速恢复 (Recovery)	68
6.5 _version	69
第 7 章 写流程	71
7.1 文档操作的定义	71
7.2 可选参数	72
7.3 Index/Bulk 基本流程	72
7.4 Index/Bulk 详细流程	73
7.4.1 协调节点流程	74
7.4.2 主分片节点流程	79
7.4.3 副分片节点流程	82
7.5 I/O 异常处理	82
7.5.1 Engine 关闭过程	83





7.5.2 Master 的对应处理	84
7.5.3 异常流程总结	84
7.6 系统特性	84
7.7 思考	85
第 8 章 GET 流程	86
8.1 可选参数	87
8.2 GET 基本流程	87
8.3 GET 详细分析	88
8.3.1 协调节点	89
8.3.2 数据节点	91
8.4 MGET 流程分析	93
8.5 思考	94
第 9 章 Search 流程	95
9.1 索引和搜索	96
9.1.1 建立索引	97
9.1.2 执行搜索	97
9.2 search type	98
9.3 分布式搜索过程	98
9.3.1 协调节点流程	99
9.3.2 执行搜索的数据节点流程	106
9.4 小结	108
第 10 章 索引恢复流程分析	110
10.1 相关配置	111
10.2 流程概述	111
10.3 主分片恢复流程	112
10.4 副分片恢复流程	116
10.4.1 流程概述	116
10.4.2 synced flush 机制	118
10.4.3 副分片节点处理过程	118
10.4.4 主分片节点处理过程	123
10.5 recovery 速度优化	127
10.6 如何保证副分片和主分片一致	128
10.7 recovery 相关监控命令	131



10.8 小结	133
第 11 章 gateway 模块分析	134
11.1 元数据	134
11.2 元数据的持久化	135
11.3 元数据的恢复	136
11.4 元数据恢复流程分析	137
11.4.1 选举集群级和索引级别的元数据	138
11.4.2 触发 allocation	140
11.5 思考	140
第 12 章 allocation 模块分析	141
12.1 什么是 allocation	141
12.2 触发时机	142
12.3 allocation 模块结构概述	142
12.4 allocators	142
12.5 deciders	143
12.5.1 负载均衡类	144
12.5.2 并发控制类	145
12.5.3 条件限制类	145
12.6 核心 reroute 实现	146
12.6.1 集群启动时 reroute 的触发时机	147
12.6.2 流程分析	147
12.6.3 gatewayAllocator	147
12.6.4 shardsAllocator	154
12.7 从 gateway 到 allocation 流程的转换	154
12.8 从 allocation 流程到 recovery 流程的转换	155
12.9 思考	156
第 13 章 Snapshot 模块分析	157
13.1 仓库	158
13.2 快照	160
13.2.1 创建快照	160
13.2.2 获取快照信息	161
13.2.3 快照 status	163
13.2.4 取消、删除快照和恢复操作	163

13.3	从快照恢复	164
13.3.1	部分恢复	165
13.3.2	恢复过程中更改索引设置	165
13.3.3	监控恢复进度	165
13.4	创建快照的实现原理	166
13.4.1	Lucene 文件格式简介	167
13.4.2	协调节点流程	168
13.4.3	主节点流程	170
13.4.4	数据节点流程	173
13.5	删除快照实现原理	184
13.5.1	协调节点流程	184
13.5.2	主节点流程	185
13.6	思考与总结	192
第 14 章	Cluster 模块分析	194
14.1	集群状态	194
14.2	内部封装和实现	198
14.2.1	MasterService	198
14.2.2	ClusterApplierService	199
14.2.3	线程池	201
14.3	提交集群任务	202
14.3.1	内部模块如何提交任务	203
14.3.2	任务提交过程实现	205
14.4	集群任务的执行过程	209
14.5	集群状态的发布过程	211
14.5.1	增量发布的实现原理	213
14.5.2	二段提交总流程	214
14.5.3	发布过程	215
14.5.4	提交过程	216
14.5.5	异常处理	217
14.6	应用集群状态	217
14.7	查看等待执行的集群任务	219
14.8	任务管理 API	220
14.8.1	列出运行中的任务	221
14.8.2	取消任务	222
14.9	思考与总结	222

第 15 章 Transport 模块分析	223
15.1 配置信息	223
15.1.1 传输模块配置	223
15.1.2 通用网络配置	225
15.2 Transport 总体架构	227
15.2.1 网络层	227
15.2.2 服务层	229
15.3 REST 解析和处理	234
15.4 RPC 实现	235
15.4.1 RPC 的注册和映射	236
15.4.2 根据 Action 获取处理类	240
15.5 思考与总结	241
第 16 章 ThreadPool 模块分析	242
16.1 线程池类型	243
16.1.1 fixed	244
16.1.2 scaling	244
16.1.3 direct	245
16.1.4 fixed_auto_queue_size	245
16.2 处理器设置	245
16.3 查看线程池	246
16.3.1 cat thread pool	246
16.3.2 nodes info	247
16.3.3 nodes stats	248
16.3.4 nodes hot threads	248
16.3.5 Java 的线程池结构	250
16.4 ES 的线程池实现	252
16.4.1 ThreadPool 类结构与初始化	253
16.4.2 fixed 类型线程池构建过程	255
16.4.3 scaling 类型线程池构建过程	256
16.4.4 direct 类型线程池构建过程	256
16.4.5 fixed_auto_queue_size 类型线程池构建过程	257
16.5 其他线程池	258
16.6 思考与总结	258
第 17 章 Shrink 原理分析	259

17.1	准备源索引	259
17.2	缩小索引	260
17.3	Shrink 的工作原理	260
17.3.1	创建新索引	261
17.3.2	创建硬链接	261
17.3.3	硬链接过程源码分析	262
第 18 章 写入速度优化		264
18.1	translog flush 间隔调整	264
18.2	索引刷新闻隔 refresh_interval	265
18.3	段合并优化	265
18.4	indexing buffer	266
18.5	使用 bulk 请求	267
18.5.1	bulk 线程池和队列	267
18.5.2	并发执行 bulk 请求	267
18.6	磁盘间的任务均衡	268
18.7	节点间的任务均衡	268
18.8	索引过程调整和优化	269
18.8.1	自动生成 doc ID	269
18.8.2	调整字段 Mappings	269
18.8.3	调整_source 字段	269
18.8.4	禁用_all 字段	270
18.8.5	对 Analyzed 的字段禁用 Norms	271
18.8.6	index_options 设置	271
18.9	参考配置	271
18.10	思考与总结	272
第 19 章 搜索速度的优化		273
19.1	为文件系统 cache 预留足够的内存	273
19.2	使用更快的硬件	273
19.3	文档模型	274
19.4	预索引数据	274
19.5	字段映射	276
19.6	避免使用脚本	276
19.7	优化日期搜索	276
19.8	为只读索引执行 force-merge	278

19.9	预热全局序号 (global ordinals)	279
19.10	execution hint.....	279
19.11	预热文件系统 cache.....	280
19.12	转换查询表达式	280
19.13	调节搜索请求中的 batched_reduce_size	281
19.14	使用近似聚合	281
19.15	深度优先还是广度优先.....	281
19.16	限制搜索请求的分片数.....	281
19.17	利用自适应副本选择 (ARS) 提升 ES 响应速度	282
第 20 章	磁盘使用量优化	285
20.1	预备知识	285
20.1.1	元数据字段	285
20.1.2	索引映射参数	286
20.2	优化措施	287
20.2.1	禁用对你来说不需要的特性.....	287
20.2.2	禁用 doc values	290
20.2.3	不要使用默认的动态字符串映射.....	290
20.2.4	观察分片大小	291
20.2.5	禁用 _source.....	291
20.2.6	使用 best_compression	291
20.2.7	Force Merge.....	292
20.2.8	Shrink Index	292
20.2.9	数值类型长度够用就好	292
20.2.10	使用索引排序来排列类似的文档.....	292
20.2.11	在文档中以相同的顺序放置字段.....	292
20.3	测试数据	293
第 21 章	综合应用实践.....	294
21.1	集群层	294
21.1.1	规划集群规模	294
21.1.2	单节点还是多节点部署	295
21.1.3	移除节点	295
21.1.4	独立部署主节点	296
21.2	节点层	296
21.2.1	控制线程池的队列大小	296

21.2.2 为系统 cache 保留一半物理内存	297
21.3 系统层	297
21.3.1 关闭 swap	297
21.3.2 配置 Linux OOM Killer	297
21.3.3 优化内核参数	298
21.4 索引层	304
21.4.1 使用全局模板	304
21.4.2 索引轮转	304
21.4.3 避免热索引分片不均	305
21.4.4 副本数选择	306
21.4.5 Force Merge	306
21.4.6 Shrink Index	306
21.4.7 close 索引	307
21.4.8 延迟分配分片	307
21.4.9 小心地使用 fielddata	307
21.5 客户端	308
21.5.1 使用 REST API 而非 Java API	308
21.5.2 注意 429 状态码	308
21.5.3 curl 的 HEAD 请求	308
21.5.4 了解你的搜索计划	309
21.5.5 为读写请求设置比较长的超时时间	309
21.6 读写	309
21.6.1 避免搜索操作返回巨大的结果集	309
21.6.2 避免索引巨大的文档	309
21.6.3 避免使用多个 _type	310
21.6.4 避免使用 _all 字段	310
21.6.5 避免将请求发送到同一个协调节点	310
21.7 控制相关度	311
第 22 章 故障诊断	316
22.1 使用 Profile API 定位慢查询	317
22.2 使用 Explain API 分析未分配的分片 (Unassigned Shards)	320
22.2.1 诊断未分配的主分片	320
22.2.2 诊断未分配的副分片	324
22.2.3 诊断已分配的分片	326
22.3 节点 CPU 使用率高	328

22.4	节点内存使用率高	330
22.5	Slow Logs	333
22.6	分析工具	334
22.6.1	I/O 信息	334
22.6.2	内存	335
22.6.3	CPU 信息	337
22.6.4	网络连接和流量	339
22.7	小结	341
附录 A	重大版本变化	342



第 1 章

走进 Elasticsearch

本章主要介绍入门知识，对 Elasticsearch 基本概念已经很熟悉的读者可以跳过本章。

1.1 基本概念和原理

Elasticsearch 是实时的分布式搜索分析引擎，内部使用 Lucene 做索引与搜索。

何谓实时？新增到 ES 中的数据在 1 秒后就可以被检索到，这种新增数据对搜索的可见性称为“准实时搜索”。分布式意味着可以动态调整集群规模，弹性扩容，而这一切操作起来都非常简便，用户甚至不必了解集群原理就可以实现。按官方的描述，集群规模支持“上百”个节点，相比 HDFS 等上千台的集群，这个规模“小了点”。影响集群规模上限的原因将在后续的章节中分析。因此，目前我们认为 ES 适合中等数据量的业务，不适合存储海量数据。

Lucene 是 Java 语言编写的全文搜索框架，用于处理纯文本的数据，但它只是一个库，提供建立索引、执行搜索等接口，但不包含分布式服务，这些正是 ES 做的。什么是全文？对全部的文本内容进行分析，建立索引，使之可以被搜索，因此称为全文。

基于 ES，你可以很容易地搭建自己的搜索引擎，用于分析日志，或者配合开源爬虫建立某个垂直领域的搜索引擎。ES 易用的产品设计使得它很容易上手。除了搜索，ES 还提供了大量的聚合功能，所以它不单单是一个搜索引擎，还可以进行数据分析、统计，生成指标数据。而这些功能都在快速迭代，目前每 2 周左右就会发布新版本。

1.1.1 索引结构

ES 是面向文档的。各种文本内容以文档的形式存储到 ES 中，文档可以是一封邮件、一条日志，或者一个网页的内容。一般使用 JSON 作为文档的序列化格式，文档可以有很多字段，在创建索引的时候，我们需要描述文档中每个字段的数据类型，并且可能需要指定不同的分析器，就像在关系型数据中“CREATE TABLE”一样。

在存储结构上，由 `_index`、`_type` 和 `_id` 唯一标识一个文档。

`_index` 指向一个或多个物理分片的逻辑命名空间，`_type` 类型用于区分同一个集合中的不同细分，在不同的细分中，数据的整体模式是相同或相似的，不适合完全不同类型的数据。多个 `_type` 可以在相同的索引中存在，只要它们的字段不冲突即可（对于整个索引，映射在本质上被“扁平化”成一个单一的、全局的模式）。`_id` 文档标记符由系统自动生成或使用者提供。

很多初学者喜欢套用 RDBMS 中的概念，将 `_index` 理解为数据库，将 `_type` 理解为表，这是很牵强的理解，实际上这是完全不同的概念，没什么相似性，不同 `_type` 下的字段不能冲突，删除整个 `_type` 也不会释放空间。在实际应用中，数据模型不同，有不同 `_type` 需求的时候，我们应该建立单独的索引，而不是在一个索引下使用不同的 `_type`。删除过期老化的数据时，最好以索引为单位，而不是 `_type` 和 `_id`。正由于 `_type` 在实际应用中容易引起概念混淆，以及允许索引存在多 `_type` 并没有什么实际意义，在 ES 6.x 版本中，一个索引只允许存在一个 `_type`，未来的 7.x 版本将完全删除 `_type` 的概念。

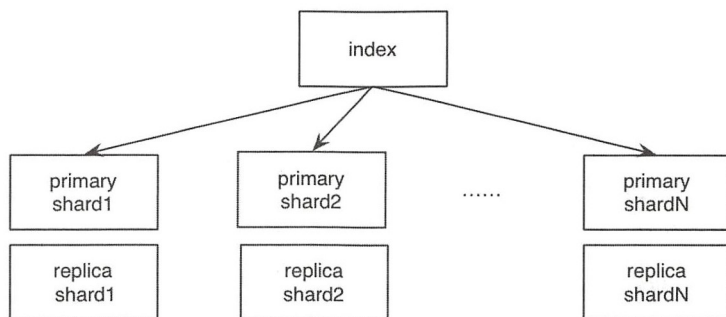
1.1.2 分片 (shard)

在分布式系统中，单机无法存储规模巨大的数据，要依靠大规模集群处理和存储这些数据，一般通过增加机器数量来提高系统水平扩展能力。因此，需要将数据分成若干小块分配到各个机器上。然后通过某种路由策略找到某个数据块所在的位置。

除了将数据分片以提高水平扩展能力，分布式存储中还会把数据复制成多个副本，放置到不同的机器中，这样一来可以增加系统可用性，同时数据副本还可以使读操作并发执行，分担集群压力。但是多数据副本也带来了一致性的问题：部分副本写成功，部分副本写失败。我们随后讨论。

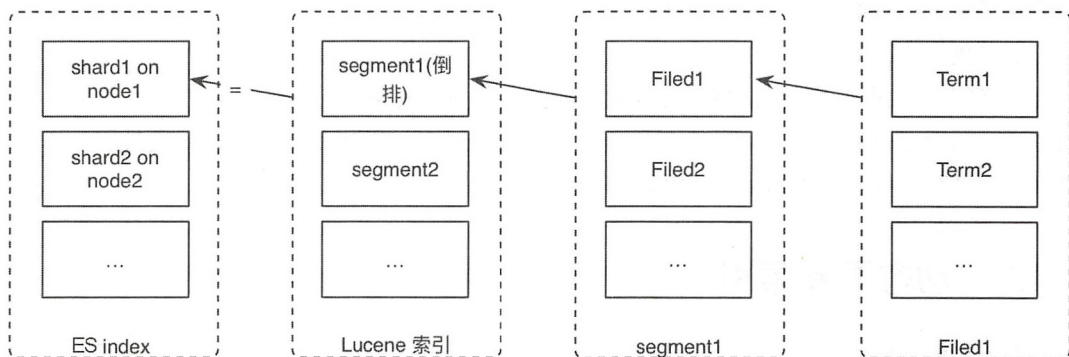
为了应对并发更新问题，ES 将数据副本分为主从两部分，即主分片（primary shard）和副分片（replica shard）。主数据作为权威数据，写过程中先写主分片，成功后再写副分片，恢复阶段以主分片为准。

数据分片和数据副本的关系如下图所示。



分片 (shard) 是底层的基本读写单元, 分片的目的是分割巨大索引, 让读写可以并行操作, 由多台机器共同完成。读写请求最终落到某个分片上, 分片可以独立执行读写工作。ES 利用分片将数据分发到集群内各处。分片是数据的容器, 文档保存在分片内, 不会跨分片存储。分片又被分配到集群内的各个节点里。当集群规模扩大或缩小时, ES 会自动在各节点中迁移分片, 使数据仍然均匀分布在集群里。

索引与分片的关系如下图所示。



一个 ES 索引包含很多分片, 一个分片是一个 Lucene 的索引, 它本身就是一个完整的搜索引擎, 可以独立执行建立索引和搜索任务。Lucene 索引又由很多分段组成, 每个分段都是一个倒排索引。ES 每次 “refresh” 都会生成一个新的分段, 其中包含若干文档的数据。在每个分段内部, 文档的不同字段被单独建立索引。每个字段的值由若干词 (Term) 组成, Term 是原文本内容经过分词器处理和语言处理后的最终结果 (例如, 去除标点符号和转换为词根)。

如果想了解 Lucene 分段由哪些文件组成, 每个文件都存储了什么内容, 则可以参考 Apache Lucene 的手册: http://lucene.apache.org/core/7_3_0/core/org/apache/lucene/codecs/lucene70/package-summary.html#package.description。

索引建立的时候就需要确定好主分片数, 在较老的版本中 (5.x 版本之前), 主分片数量不

可以修改，副分片数可以随时修改。现在（5.x~6.x 版本之后），ES 已经支持在一定条件的限制下，对某个索引的主分片进行拆分（Split）或缩小（Shrink）。但是，我们仍然需要在一开始就尽量规划好主分片数量：先依据硬件情况定好单个分片容量，然后依据业务场景预估数据量和增长量，再除以单个分片容量。

分片数不够时，可以考虑新建索引，搜索 1 个有着 50 个分片的索引与搜索 50 个每个都有 1 个分片的索引完全等价，或者使用 `_split` API 来拆分索引（6.1 版本开始支持）。

在实际应用中，我们不应该向单个索引持续写数据，直到它的分片巨大无比。巨大的索引会在数据老化后难以删除，以 `_id` 为单位删除文档不会立刻释放空间，删除的 `doc` 只在 Lucene 分段合并时才会真正从磁盘删除。即使手工触发分段合并，仍然会引起较高的 I/O 压力，并且可能因为分段巨大导致在合并过程中磁盘空间不足（分段大小大于磁盘可用空间的一半）。因此，我们建议周期性地创建新索引。例如，每天创建一个。假如有一个索引 `website`，可以将它命名为 `website_20180319`。然后创建一个名为 `website` 的索引别名来关联这些索引。这样，对于业务方来说，读取时使用的名称不变，当需要删除数据的时候，可以直接删除整个索引。

索引别名就像一个快捷方式或软链接，不同的是它可以指向一个或多个索引。可以用于实现索引分组，或者索引间的无缝切换。

现在我们已经确定好了主分片数量，并且保证单个索引的数据量不会太大，周期性创建新索引带来的一个新问题是集群整体分片数量较多，集群管理的总分片数越多压力就越大。在每天生成一个新索引的场景中，可能某天产生的数据量很小，实际上不需要这么多分片，甚至一个就够。这时，可以使用 `_shrink` API 来缩减主分片数量，降低集群负载。

1.1.3 动态更新索引

为文档建立索引，使其每个字段都可以被搜索，通过关键词检索文档内容，会使用倒排索引的数据结构。倒排索引一旦被写入文件后就具有不变性，不变性具有许多好处：对文件的访问不需要加锁，读取索引时可以被文件系统缓存等。

那么索引如何更新，让新添加的文档可以被搜索到？答案是使用更多的索引，新增内容并写到一个新的倒排索引中，查询时，每个倒排索引都被轮流查询，查询完再对结果进行合并。

每次内存缓冲的数据被写入文件时，会产生一个新的 Lucene 段，每个段都是一个倒排索引。在一个记录元信息的文件中描述了当前 Lucene 索引都含有哪些分段。

由于分段的不变性，更新、删除等操作实际上是将数据标记为删除，记录到单独的位置，这种方式称为标记删除。因此删除部分数据不会释放磁盘空间。

1.1.4 近实时搜索

在写操作中，一般会先在内存中缓冲一段数据，再将这些数据写入硬盘，每次写入硬盘的这批数据称为一个分段，如同任何写操作一样。一般情况下（direct 方式除外），通过操作系统 write 接口写到磁盘的数据先到达系统缓存（内存），write 函数返回成功时，数据未必被刷到磁盘。通过手工调用 flush，或者操作系统通过一定策略将系统缓存刷到磁盘。这种策略大幅提升了写入效率。从 write 函数返回成功开始，无论数据有没有被刷到磁盘，该数据已经对读取可见。

ES 正是利用这种特性实现了近实时搜索。每秒产生一个新分段，新段先写入文件系统缓存，但稍后再执行 flush 刷盘操作，写操作很快会执行完，一旦写成功，就可以像其他文件一样被打开和读取了。

由于系统先缓冲一段数据才写，且新段不会立即刷入磁盘，这两个过程中如果出现某些意外情况（如主机断电），则会存在丢失数据的风险。通用的做法是记录事务日志，每次对 ES 进行操作时均记录事务日志，当 ES 启动的时候，重放 translog 中所有在最后一次提交后发生的变更操作。比如 HBase 等都有自己的事务日志。

1.1.5 段合并

在 ES 中，每秒清空一次写缓冲，将这些数据写入文件，这个过程称为 refresh，每次 refresh 会创建一个新的 Lucene 段。但是分段数量太多会带来较大的麻烦，每个段都会消耗文件句柄、内存。每个搜索请求都需要轮流检查每个段，查询完再对结果进行合并；所以段越多，搜索也就越慢。因此需要通过一定的策略将这些较小的段合并为大的段，常用的方案是选择大小相似的分段进行合并。在合并过程中，标记为删除的数据不会写入新分段，当合并过程结束，旧的分段数据被删除，标记删除的数据才从磁盘删除。

HBase、Cassandra 等系统都有类似的分段机制，写过程中先在内存缓冲一批数据，不时地将这些数据写入文件作为一个分段，分段具有不变性，再通过一些策略合并分段。分段合并过程中，新段的产生需要一定的磁盘空间，我们要保证系统有足够的剩余可用空间。Cassandra 系统在段合并过程中的一个问题就是，当持续地向一个表中写入数据，如果段文件大小没有上限，当巨大的段达到磁盘空间的一半时，剩余空间不足以进行新的段合并过程。如果段文件设置一定上限不再合并，则对表中部分数据无法实现真正的物理删除。ES 存在同样的问题。

1.2 集群内部原理

分布式系统的集群方式大致可以分为主从（Master-Slave）模式和无主模式。ES、HDFS、HBase 使用主从模式，Cassandra 使用无主模式。主从模式可以简化系统设计，Master 作为权威节点，部分操作仅由 Master 执行，并负责维护集群元信息。缺点是 Master 节点存在单点故障，需要解决灾备问题，并且集群规模会受限于 Master 节点的管理能力。

因此，从集群节点角色的角度划分，至少存在主节点和数据节点，另外还有协调节点、预处理节点和部落节点，下面分别介绍各种类型节点的职能。

1.2.1 集群节点角色

1. 主节点（Master node）

主节点负责集群层面的相关操作，管理集群变更。

通过配置 `node.master: true`（默认）使节点具有被选举为 Master 的资格。主节点是全局唯一的，将从有资格成为 Master 的节点中进行选举。

主节点也可以作为数据节点，但尽可能做少量的工作，因此生产环境应尽量分离主节点和数据节点，创建独立主节点的配置：

```
node.master: true
node.data: false
```

为了防止数据丢失，每个主节点应该知道有资格成为主节点的数量，默认为 1，为避免网络分区时出现多主的情况，配置 `discovery.zen.minimum_master_nodes` 原则上最小值应该是：

$$(\text{master_eligible_nodes} / 2) + 1$$

2. 数据节点（Data node）

负责保存数据、执行数据相关操作：CRUD、搜索、聚合等。数据节点对 CPU、内存、I/O 要求较高。一般情况下（有一些例外，后续章节会给出），数据读写流程只和数据节点交互，不会和主节点打交道（异常情况除外）。

通过配置 `node.data: true`（默认）来使一个节点成为数据节点，也可以通过下面的配置创建一个数据节点：

```
node.master: false
node.data: true
node.ingest: false
```

3. 预处理节点 (Ingest node)

这是从 5.0 版本开始引入的概念。预处理操作允许在索引文档之前，即写入数据之前，通过事先定义好的一系列的 processors (处理器) 和 pipeline (管道)，对数据进行某种转换、富化。processors 和 pipeline 拦截 bulk 和 index 请求，在应用相关操作后将文档传回给 index 或 bulk API。

默认情况下，在所有的节点上启用 ingest，如果想在某个节点上禁用 ingest，则可以添加配置 node.ingest: false，也可以通过下面的配置创建一个仅用于预处理的节点：

```
node.master: false
node.data: false
node.ingest: true
```

4. 协调节点 (Coordinating node)

客户端请求可以发送到集群的任何节点，每个节点都知道任意文档所处的位置，然后转发这些请求，收集数据并返回给客户端，处理客户端请求的节点称为协调节点。

协调节点将请求转发给保存数据的数据节点。每个数据节点在本地执行请求，并将结果返回协调节点。协调节点收集完数据后，将每个数据节点的结果合并为单个全局结果。对结果收集和排序的过程可能需要很多 CPU 和内存资源。

通过下面的配置创建一个仅用于协调的节点：

```
node.master: false
node.data: false
node.ingest: false
```

5. 部落节点 (Tribe node)

tribes (部落) 功能允许部落节点在多个集群之间充当联合客户端。

在 ES 5.0 之前还有一个客户端节点 (Node Client) 的角色，客户端节点有以下属性：

```
node.master: false
node.data: false
```

它不做主节点，也不做数据节点，仅用于路由请求，本质上是一个智能负载均衡器（从负载均衡器的定义来说，智能和非智能的区别在于是否知道访问的内容存在于哪个节点），从 5.0 版本开始，这个角色被协调节点（Coordinating only node）取代。

1.2.2 集群健康状态

从数据完整性的角度划分，集群健康状态分为三种：

- Green，所有的主分片和副分片都正常运行。
- Yellow，所有的主分片都正常运行，但不是所有的副分片都正常运行。这意味着存在单点故障风险。
- Red，有主分片没能正常运行。

每个索引也有上述三种状态，假设丢失了一个副分片，该分片所属的索引和整个集群变为 Yellow 状态，其他索引仍为 Green。

1.2.3 集群状态

集群状态元数据是全局信息，元数据包括内容路由信息、配置信息等，其中最重要的是内容路由信息，它描述了“哪个分片位于哪个节点”这种信息。

集群状态由主节点负责维护，如果主节点从数据节点接收更新，则将这些更新广播到集群的其他节点，让每个节点上的集群状态保持最新。ES 2.0 版本之后，更新的集群状态信息只发增量内容，并且是被压缩的。

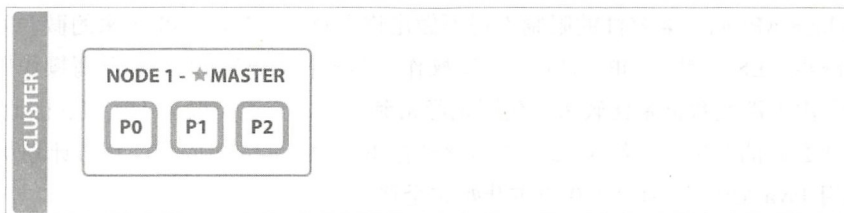
1.2.4 集群扩容

当扩容集群、添加节点时，分片会均衡地分配到集群的各个节点，从而对索引和搜索过程进行负载均衡，这些都是系统自动完成的。

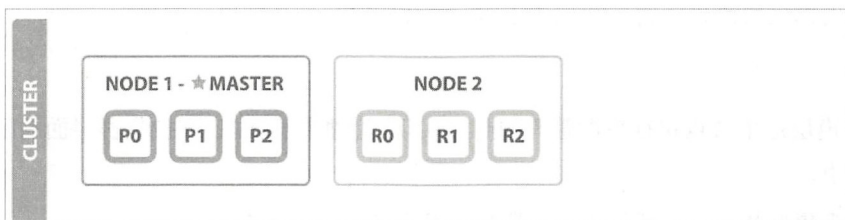
分片副本实现了数据冗余，从而防止硬件故障导致的数据丢失。

下面演示了当集群只有一个节点，到变成两个节点、三个节点时的 shard 迁移过程示例（图片来自官网）。

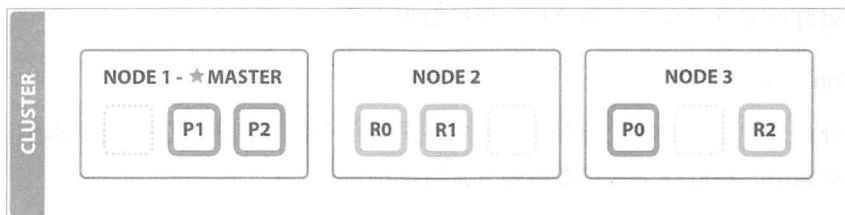
起初，在 NODE1 上有三个主分片，没有副分片，如下图所示。



其中，P 代表 Primary shard；R 代表 Replica shard。以后出现的内容使用相同的简称。
添加第二个节点后，副分片被分配到 NODE2，如下图所示。



添加第三个节点后，索引的六个分片被平均分配到集群的三个节点，如下图所示。



分片分配过程中除了让节点间均匀存储，还要保证不把主分片和副分片分配到同一节点，避免单个节点故障引起数据丢失。

分布式系统中难免出现故障，当节点异常时，ES 会自动处理节点异常。当主节点异常时，集群会重新选举主节点。当某个主分片异常时，会将副分片提升为主分片。

1.3 客户端 API

当需要实现一个客户端对集群进行读写操作时，可以选择 REST 接口、Java REST API，或者 Java API。

Java REST API 是对原生 REST 接口的封装。REST 接口、Java REST API 使用 9200 端口通信，采用 JSON over HTTP 方式，Java API 使用 9300 端口通信，数据序列化为二进制。

使用 Java API 理论上来说效率更高一些，但是后来官方发现实际上相差无几，但是版本迭

代中却因为 Java API 向下兼容性的限制不得不做出许多牺牲，Java API 带来的微弱效率优势远不及带来的缺点。ES 不是高 QPS 的应用，写操作非常消耗 CPU 资源，因此写操作属于比较长的操作，聚合由于涉及数据量比较大，延迟也经常到秒级，查询一般也不密集。因此 RPC 框架的效率没有那么高的要求。后续 Java API 将逐渐被 Java REST API 取代。官方计划从 ES 7.0 开始不建议使用 Java API，并且从 8.0 版本开始完全移除。

1.4 主要内部模块简介

在分析内部模块流程之前，我们先了解一下 ES 中几个基础模块的功能。

Cluster

Cluster 模块是主节点执行集群管理的封装实现，管理集群状态，维护集群层面的配置信息。

主要功能如下：

- 管理集群状态，将新生成的集群状态发布到集群所有节点。
- 调用 allocation 模块执行分片分配，决策哪些分片应该分配到哪个节点
- 在集群各节点中直接迁移分片，保持数据平衡。

allocation

封装了分片分配相关的功能和策略，包括主分片的分配和副分片的分配，本模块由主节点调用。创建新索引、集群完全重启都需要分片分配的过程。

Discovery

发现模块负责发现集群中的节点，以及选举主节点。当节点加入或退出集群时，主节点会采取相应的行动。从某种角度来说，发现模块起到类似 ZooKeeper 的作用，选主并管理集群拓扑。

gateway

负责对收到 Master 广播下来的集群状态（cluster state）数据的持久化存储，并在集群完全重启时恢复它们。

Indices

索引模块管理全局级的索引设置，不包括索引级的（索引设置分为全局级和每个索引级）。它还封装了索引数据恢复功能。集群启动阶段需要的主分片恢复和副分片恢复就是在这个模块实现的。

HTTP

HTTP 模块允许通过 JSON over HTTP 的方式访问 ES 的 API，HTTP 模块本质上是完全异步的，这意味着没有阻塞线程等待响应。使用异步通信进行 HTTP 的好处是解决了 C10k 问题（10k 量级的并发连接）。

在部分场景下，可考虑使用 HTTP keepalive 以提升性能。注意：不要在客户端使用 HTTP chunking。

Transport

传输模块用于集群内节点之间的内部通信。从一个节点到另一个节点的每个请求都使用传输模块。

如同 HTTP 模块，传输模块本质上也是完全异步的。

传输模块使用 TCP 通信，每个节点都与其他节点维持若干 TCP 长连接。内部节点间的所有通信都是本模块承载的。

Engine

Engine 模块封装了对 Lucene 的操作及 translog 的调用，它是对一个分片读写操作的最终提供者。

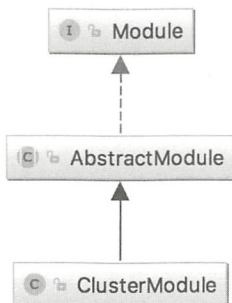
ES 使用 Guice 框架进行模块化管理。Guice 是 Google 开发的轻量级依赖注入框架（IoC）。

软件设计中经常说要依赖于抽象而不是具象，IoC 就是这种理念的实现方式，并且在内部实现了对象的创建和管理。

1.4.1 模块结构

在 Guice 框架下，一个典型的模块由 Service 和 Module 类（类名可以自由定义）组成，Service 用于实现业务功能，Module 类中配置绑定信息。

以 ClusterModule 为例，类结构如下图所示。



`AbstractModule` 是 Guice 提供的基类，模块需要从这个类继承。`Module` 类的主要作用是定义绑定关系，例如：

```
protected void configure() {  
    //绑定实现类  
    bind(ClusterService.class).toInstance(clusterService);  
}
```

1.4.2 模块管理

定义好的模块由 `ModulesBuilder` 类统一管理，`ModulesBuilder` 是 ES 对 Guice 的封装，内部调用 Guice 接口，主要对外提供两个方法。

- `add` 方法：添加创建好的模块
- `createInjector` 方法：调用 `Guice.createInjector` 创建并返回 `Injector`，后续通过 `Injector` 获取相应 `Service` 类的实例。

使用 `ModulesBuilder` 进行模块管理的伪代码示例：

```
ModulesBuilder modules = new ModulesBuilder();  
  
//以 Cluster 模块为例  
ClusterModule clusterModule = new ClusterModule();  
modules.add(clusterModule);  
  
//省略其他模块的创建和添加  
...  
  
//创建 Injector，并获取相应类的实例  
injector = modules.createInjector();  
setGatewayAllocator(injector.getInstance(GatewayAllocator.class))
```

模块化的封装让 ES 易于扩展，插件本身也是一个模块，节点启动时被模块管理器添加进来。



2 chapter

第 2 章 准备编译和调试环境

2.1 编译源码

2.1.1 准备 JDK 和 Gradle

Elasticsearch 是 Java 语言编写的。运行和编译 Elasticsearch 时，对 JDK 版本的选择请参考手册，地址：https://www.elastic.co/guide/en/elasticsearch/reference/current/_installation.html。推荐使用 Oracle 的 JDK 版本。本书使用的 JDK 版本为 1.8.0_121。JDK 的安装方式不在这里讨论，请读者参考 Oracle 官方网站。

从 Elasticsearch 5.0 开始，构建工具由 Maven 更改为 Gradle，本书使用的 Gradle 版本为 4.6，安装方式请参考官方网站。

2.1.2 下载源代码

Elasticsearch 的代码托管于 GitHub，可以“git clone”最新的 Master 分支或某个 tag，也可以到 Releases 或 Tags 下载各个版本的代码包：<https://github.com/elastic/elasticsearch/releases>。

解压源码包到自己的目录，以 v6.1.2 版本为例：

```
tar -xzvf elasticsearch-6.1.2.tar.gz
```



2.1.3 编译项目，打包

本书以 Linux 平台编译为例。切换到解压后的源码目录，执行编译：

```
cd elasticsearch-6.1.2
./gradlew assemble
```

编译完成后，系统打印“BUILD SUCCESSFUL”代表编译成功。完整的打包文件位于./distribution 目录，包含 tar、zip 等多种格式的包。以 tar 为例，包路径为./distribution/tar/build/distributions/ elasticsearch-6.1.2-SNAPSHOT.tar.gz。默认带有“SNAPSHOT”后缀，如果想去掉“SNAPSHOT”，则可以在编译时添加-Dbuild.snapshot=false，即：

```
./gradlew assemble -Dbuild.snapshot=false
```

Elasticsearch 的版本号由四位组成，格式为：

主版本.次版本.修正版本.构建版本

但是最后一位“构建版本”不出现在构建后的包名称中。默认情况下每个号码均不会大于100。当完成对代码的修改，编译、发布时要修改版本号，添加自己的版本号，可以在原修正版本或构建版本的基础上增加。与修改版本号相关的两个文件：buildSrc/version.properties 和./core/src/main/java/org/elasticsearch/Version.java。文件位置可能因版本不同而有所不同。

2.1.4 将工程导入 IntelliJ IDEA

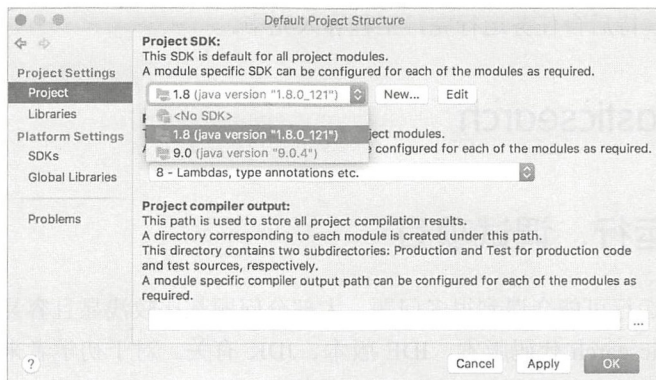
本书使用的 IDE 以 IntelliJ IDEA 2017.3（后续简称 IDEA）为例。进入 Elasticsearch 源码根目录，执行：

```
gradle idea
```

生成 IntelliJ 的项目文件，正常完成会显示“BUILD SUCCESSFUL”。

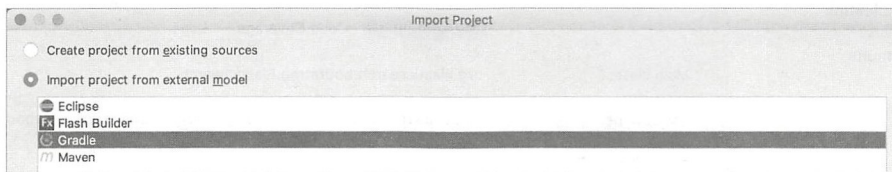
首先配置 SDK。打开 IntelliJ IDEA，在欢迎界面右下角选择“Configure→Project Defaults→Project Structure”，弹出如下图所示的界面。



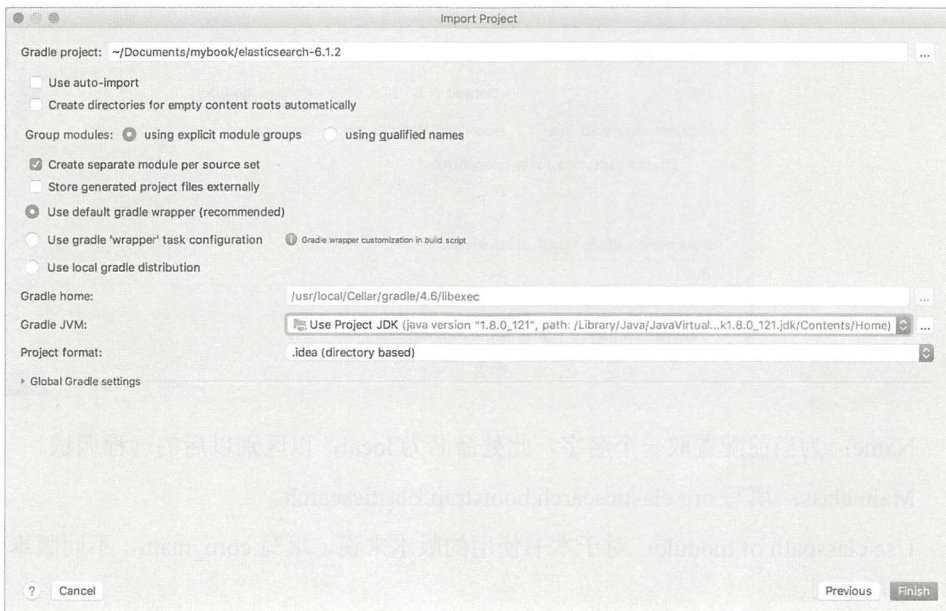


在 Project SDK 下拉列表中确认 JDK 的版本，必要时选择 “New→JDK”，选择 JDK Home 目录，添加正确的 JDK。

回到 IntelliJ IDEA 欢迎界面，选择 “Import Project”，选择 Elasticsearch-6.1.2 目录，单击 “Open”，在 Import Project 界面选择 “Gradle”，如下图所示。



单击 “Next”，设置工程名称，并确认 Grande home 和 Grande JVM 配置正确，如下图所示。



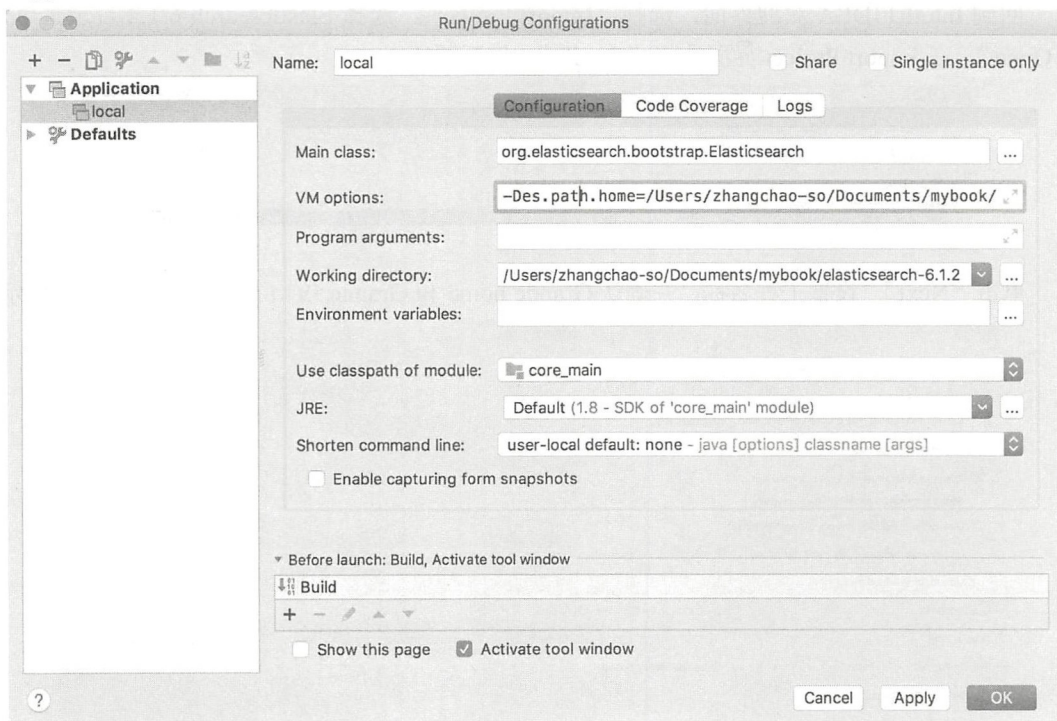
单击完成按钮，待后台任务运行完，工程导入完毕。

2.2 调试 Elasticsearch

2.2.1 本地运行、调试项目

在 IDE 环境中运行可能会遇到很多问题，大部分问题都比较浅显且容易处理，有一些较为罕见的问题和 Elasticsearch 代码版本、IDE 版本、JDK 有关。对于初学者来说，不要在这些问题上占用太长时间，否则容易让人丧失学习的动力。

在 IntelliJ IDEA 菜单中选择“Run→Edit Configurations”，单击左侧的加号“+”，选择“Application”，在弹出的配置界面中填写相关配置，如下图所示。



- **Name:** 为当前配置取一个名字，此处命名为 local，以区别以后的远程调试。
- **Main class:** 填写 org.elasticsearch.bootstrap.Elasticsearch。
- **Use classpath of module:** 对于本书使用的版本来说，填写 core_main，不同版本可能会有所不同。



- VM options: 填写内容为

```
-Des.path.home=/Users/zhangchao-so/Documents/mybook/eshome -Des.path.conf=
/Users/zhangchao-so/Documents/mybook/eshome/config -Xms1g -Xmx1g -Dlog4j2.
disable.jmx=true -Djava.security.policy=/Users/zhangchao-so/Documents/mybook/
eshome/config/elasticsearch.policy
```

在解释上述选项之前，我们需要先为调试环境准备运行时要用的 eshome 目录，Elasticsearch 需要从其中加载模块、读取配置，写入数据和日志。在你的环境上选择一个位置建立 eshome 目录，名称可以任意。然后复制必要的模块和配置文件到该目录中，可以选择从对应版本的官方二进制包中复制，也可以将上一步编译打包的软件包解压，本书使用后者。切换到源码根目录，执行：

```
cd distribution/zip/build/distributions/
unzip elasticsearch-6.1.2.zip
cd elasticsearch-6.1.2
cp -r config modules plugins /Users/zhangchao-so/Documents/mybook/eshome
```

复制完上述文件之后，现在看一下 VM options 选项，如下表所示。

参 数	说 明
-Des.path.home	指定 eshome 目录所在路径名
-Des.path.conf	指定配置文件所在路径名，后续调整节点配置时修改其下的配置文件
-Xms1g	设置 JVM 初始堆内存大小为 1GB
-Xmx1g	设置 JVM 最大允许分配的堆内存为 1GB
-Dlog4j2.disable.jmx=true	这个是 jvm.options 文件中的默认配置，不加会报“access denied”等错误。但在 IDE 中启动 Elasticsearch 不会加载 jvm.options 文件，这个文件是在 Elasticsearch 的启动脚本 elasticsearch、elasticsearch.bat 中加载的
-Djava.security.policy	同样是权限问题（不配置的话部分环境上会遇到 access denied 问题），配置 policy 文件路径，policy 文件内容如下： <pre>grant{ permission javax.management.MBeanTrustPermission "register"; permission javax.management.MBeanServerPermission "createMBeanServer"; permission java.lang.RuntimePermission "createClassLoader"; };</pre>

现在可以单击“运行”或“调试”按钮让程序在 IDE 中运行起来了！ES 可以单节点运行，

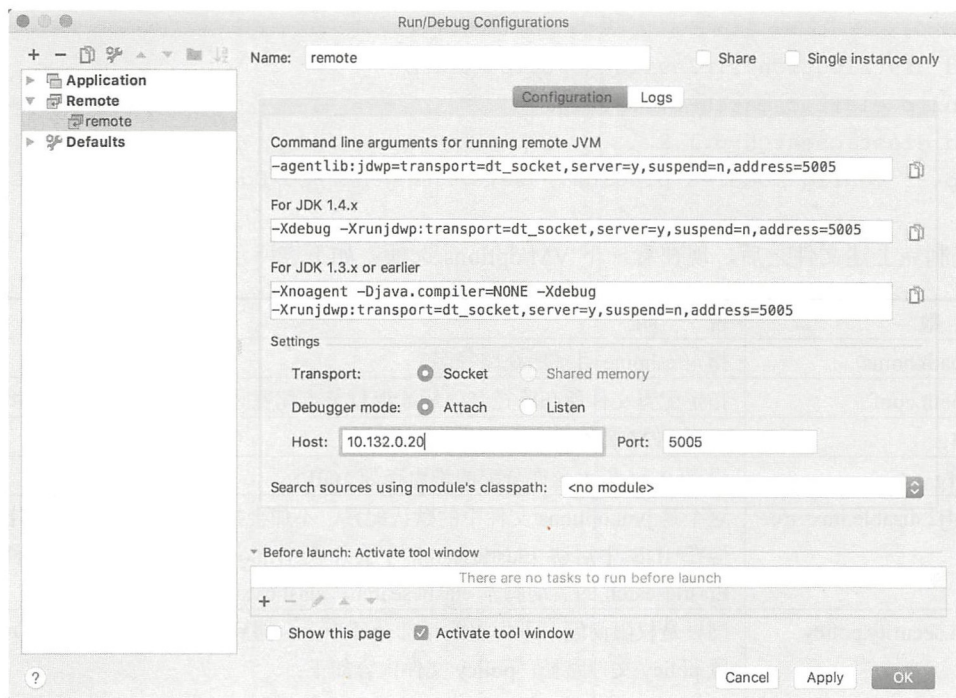


暂时不用调节任何配置。

2.2.2 远程调试

除了在本地环境调试 ES，有时需要远程调试某个节点。分布式系统总会遇到很多千奇百怪的问题，很多时候问题无法复现，或者某种问题只出现在特定环境中。在排查问题的过程中，除了查看重要的日志，远程调试节点也是一种不错的手段。

远程调试时，需要保证本地代码与远程环境运行的代码版本一致。在 IDEA 菜单中重新单击“Run→Edit Configurations”，单击左侧的加号“+”，选择“Remote”，弹出如下图所示的配置界面。



配置名称此处命名为 remote，复制“Command line arguments for running remote JVM”下的 JVM 参数到远程节点的 jvm.options 文件中：

```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005
```

重启远程节点，成功后日志信息中会有“Listening for transport dt_socket at address: 5005”信息。在 Host 中填写远程调试对象的 IP 地址，单击“OK”按钮。

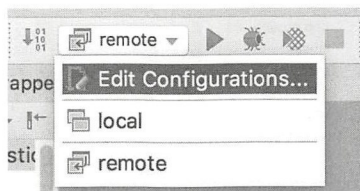


单击 **Debug** 启动调试，控制台输出：

```
Connected to the target VM, address: '10.132.0.20:5005', transport: 'socket'
```

表示连接成功，接下来在本地代码上设置断点，远程节点运行到相应的逻辑就会停住，接下来就和本地调试一样。如果需要调试启动过程，则可以设置 `suspendcy`，让程序等待调试器连接后再开始执行。

之后就在 IDEA 工具栏 “Select Run/Debug Configuration” 下拉框中选择调试本地还是远程节点，如下图所示。



远程调试节点时，停止调试不会终止远程节点进程。

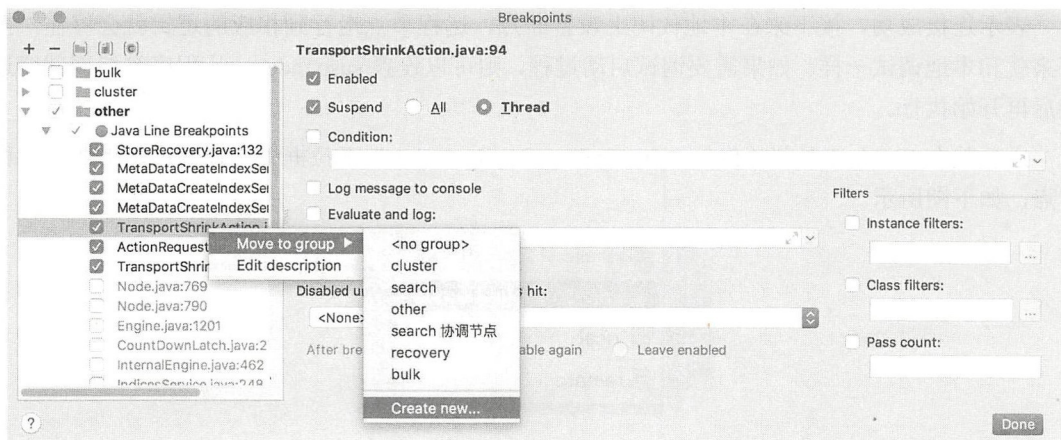
2.3 代码书签和断点组

Elasticsearch 代码量庞大，模块间关系复杂，为了更好地调试、阅读代码，我们用到 IDEA 中的两个特性：代码书签和断点组。代码书签可以标记代码位置，并为它取名。断点组可以把某个流程的一系列断点分组保存，统一开闭，例如，在调试 `recovery` 流程时，又想回头跟一下写流程，可以很方便地进行切换。

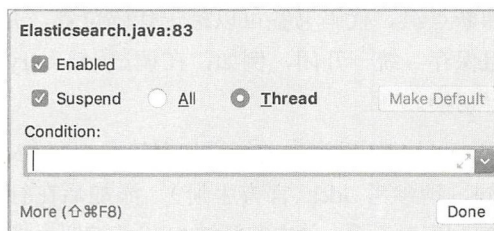
代码书签：光标选中代码行，按 `F3` 键为当前行添加书签（这里以 Mac 平台 idea 快捷键为例，不同系统的快捷键可能不同，请参考 idea 官方手册）。添加后在行号右侧会多一个对号，按 `command+F3` 组合键弹出书签列表对话框，单击左侧的 “Edit” 按钮为当前书签命名，如下图所示。



断点组：单击菜单栏“Run→View Breakpoints”，弹出断点对话框，右键单击断点，选择“Move to group”，将断点“move”到对应的分组，或者创建新组。在断点组复选框中可以批量启用/禁用整组断点，如下图所示。



在 IDEA 中断点暂停有两种策略：All 和 Thread。默认策略为 All，在这种策略下，在类的不同方法上设置两个断点，当其中一个断点被命中而暂停时，另一个线程进入第二个断点，IDEA 将不会暂停。通常在调试多线程池程序时，我们希望设置的断点，不管任何线程进入都应暂停，因此我们将策略修改为 Thread。在断点上单击右键，弹出如下图所示的对话框。



将单选按钮选中“Thread”，然后单击“Make Default”按钮，单击“Done”按钮，将其作为默认策略，但不会对之前设置的断点生效。



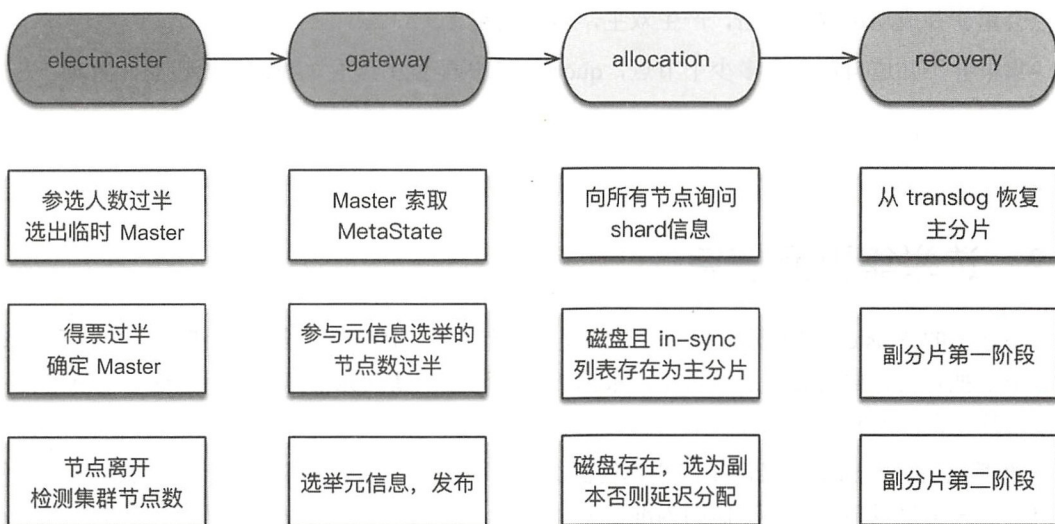
3 chapter

第 3 章 集群启动流程

让我们从启动流程开始，先在宏观上看看整个集群是如何启动的，集群状态如何从 Red 变成 Green，不涉及代码，然后分析其他模块的流程。

本书中，集群启动过程指集群完全重启时的启动过程，期间要经历选举主节点、主分片、数据恢复等重要阶段，理解其中原理和细节，对于解决或避免集群维护过程中可能遇到的脑裂、无主、恢复慢、丢数据等问题有重要作用。

集群启动的整体流程如下图所示。



3.1 选举主节点

假设有若干节点正在启动，集群启动的第一件事是从已知的活跃机器列表中选择一个作为主节点，选主之后的流程由主节点触发。

ES 的选主算法是基于 Bully 算法的改进，主要思路是对节点 ID 排序，取 ID 值最大的节点作为 Master，每个节点都运行这个流程。是不是非常简单？选主的目的是确定唯一的主节点，初学者可能认为选举出的主节点应该持有最新的元数据信息，实际上这个问题在实现上被分解为两步：先确定唯一的、大家公认的主节点，再想办法把最新的机器元数据复制到选举出的主节点上。

基于节点 ID 排序的简单选举算法有三个附加约定条件：

(1) 参选人数需要过半，达到 quorum（多数）后就选出了临时的主。为什么是临时的？每个节点运行排序取最大值的算法，结果不一定相同。举个例子，集群有 5 台主机，节点 ID 分别是 1、2、3、4、5。当产生网络分区或节点启动速度差异较大时，节点 1 看到的节点列表是 1、2、3、4，选出 4；节点 2 看到的节点列表是 2、3、4、5，选出 5。结果就不一致了，由此产生下面的第二条限制。

(2) 得票数需过半。某节点被选为主节点，必须判断加入它的节点数过半，才确认 Master 身份。解决第一个问题。

(3) 当探测到节点离开事件时，必须判断当前节点数是否过半。如果达不到 quorum，则放弃 Master 身份，重新加入集群。如果不这么做，则设想以下情况：假设 5 台机器组成的集群产生网络分区，2 台一组，3 台一组，产生分区前，Master 位于 2 台中的一个，此时 3 台一组的节点会重新并成功选取 Master，产生双主，俗称脑裂。

集群并不知道自己共有多少个节点，quorum 值从配置中读取，我们需要设置配置项：

```
discovery.zen.minimum_master_nodes
```

3.2 选举集群元信息

被选出的 Master 和集群元信息的新旧程度没有关系。因此它的第一个任务是选举元信息，让各节点把各自存储的元信息发过来，根据版本号确定最新的元信息，然后把这个信息广播下去，这样集群的所有节点都有了最新的元信息。

集群元信息的选举包括两个级别：集群级和索引级。不包含哪个 shard 存于哪个节点这种信息。这种信息以节点磁盘存储的为准，需要上报。为什么呢？因为读写流程是不经过 Master

的，Master 不知道各 shard 副本直接的数据差异。HDFS 也有类似的机制，block 信息依赖于 DataNode 的上报。

为了集群一致性，参与选举的元信息数量需要过半，Master 发布集群状态成功的规则也是等待发布成功的节点数过半。

在选举过程中，不接受新节点的加入请求。

集群元信息选举完毕后，Master 发布首次集群状态，然后开始选举 shard 级元信息。

3.3 allocation 过程

选举 shard 级元信息，构建内容路由表，是在 allocation 模块完成的。在初始阶段，所有的 shard 都处于 UNASSIGNED（未分配）状态。ES 中通过分配过程决定哪个分片位于哪个节点，重构内容路由表。此时，首先要做的是分配主分片。

1. 选主分片

现在看某个主分片[website][0]是怎么分配的。所有的分配工作都是 Master 来做的，此时，Master 不知道主分片在哪，它向集群的所有节点询问：大家把[website][0]分片的元信息发给我。然后，Master 等待所有的请求返回，正常情况下它就有了这个 shard 的信息，然后根据某种策略选一个分片作为主分片。是不是效率有些低？这种询问量=shard 数×节点数。所以说我们最好控制 shard 的总规模别太大。

现在有了 shard[website][0]的分片的多份信息，具体数量取决于副本数设置了多少。现在考虑把哪个分片作为主分片。ES 5.x 以下的版本，通过对比 shard 级元信息的版本号来决定。在多副本的情况下，考虑到如果只有一个 shard 信息汇报上来，则它一定会被选为主分片，但也许数据不是最新的，版本号比它大的那个 shard 所在节点还没启动。在解决这个问题的时候，ES 5.x 开始实施一种新的策略：给每个 shard 都设置一个 UUID，然后在集群级的元信息中记录哪个 shard 是最新的，因为 ES 是先写主分片，再由主分片节点转发请求去写副分片，所以主分片所在节点肯定是最新的，如果它转发失败了，则要求 Master 删除那个节点。所以，从 ES 5.x 开始，主分片选举过程是通过集群级元信息中记录的“最新主分片的列表”来确定主分片的：汇报信息中存在，并且这个列表中也存在。

如果集群设置了：

```
"cluster.routing.allocation.enable": "none"
```

禁止分配分片，集群仍会强制分配主分片。因此，在设置了上述选项的情况下，集群重启后的状态为 Yellow，而非 Red。

2. 选副分片

主分片选举完成后，从上一个过程汇总的 `shard` 信息中选择一个副本作为副分片。如果汇总信息中不存在，则分配一个全新副本的操作依赖于延迟配置项：

```
index.unassigned.node_left.delayed_timeout
```

我们的线上环境中最大的集群有 100+ 节点，掉节点的情况并不罕见，很多时候不能第一时间处理，这个延迟我们一般配置为以天为单位。

最后，`allocation` 过程中允许新启动的节点加入集群。

3.4 index recovery

分片分配成功后进入 `recovery` 流程。主分片的 `recovery` 不会等待其副分片分配成功才开始 `recovery`。它们是独立的流程，只是副分片的 `recovery` 需要主分片恢复完毕才开始。

为什么需要 `recovery`？对于主分片来说，可能有一些数据没来得及刷盘；对于副分片来说，一是没刷盘，二是主分片写完了，副分片还没来得及写，主副分片数据不一致。

1. 主分片 recovery

由于每次写操作都会记录事务日志（`translog`），事务日志中记录了哪种操作，以及相关的数据。因此将最后一次提交（Lucene 的一次提交就是一次 `fsync` 刷盘的过程）之后的 `translog` 中进行重放，建立 Lucene 索引，如此完成主分片的 `recovery`。

2. 副分片 recovery

副分片的恢复是比较复杂的，在 ES 的版本迭代中，副分片恢复策略有过不少调整。

副分片需要恢复成与主分片一致，同时，恢复期间允许新的索引操作。在目前的 6.0 版本中，恢复分成两阶段执行。

- **phase1**：在主分片所在节点，获取 `translog` 保留锁，从获取保留锁开始，会保留 `translog` 不受其刷盘清空的影响。然后调用 Lucene 接口把 `shard` 做快照，这是已经刷磁盘中的分片数据。把这些 `shard` 数据复制到副本节点。在 `phase1` 完毕前，会向副分片节点发送告知对方启动 `engine`，在 `phase2` 开始之前，副分片就可以正常处理写请求了。
- **phase2**：对 `translog` 做快照，这个快照里包含从 `phase1` 开始，到执行 `translog` 快照期间的新增索引。将这些 `translog` 发送到副分片所在节点进行重放。

由于需要支持恢复期间的新增写操作（让 ES 的可用性更强），这两个阶段中需要重点关注以下几个问题。

分片数据完整性：如何做到副分片不丢数据？第二阶段的 translog 快照包括第一阶段所有的新增操作。那么第一阶段执行期间如果发生“Lucene commit”（将文件系统写缓冲中的数据刷盘，并清空 translog），清除 translog 怎么办？在 ES 2.0 之前，是阻止了刷新操作，以此让 translog 都保留下来。从 2.0 版本开始，为了避免这种做法产生过大的 translog，引入了 translog.view 的概念，创建 view 可以获取后续的所有操作。从 6.0 版本开始，translog.view 被移除。引入 TranslogDeletionPolicy 的概念，它将 translog 做一个快照来保持 translog 不被清理。这样实现了在第一阶段允许 Lucene commit。

数据一致性：在 ES 2.0 之前，副分片恢复过程有三个阶段，第三阶段会阻塞新的索引操作，传输第二阶段执行期间新增的 translog，这个时间很短。自 2.0 版本之后，第三阶段被删除，恢复期间没有任何写阻塞过程。在副分片节点，重放 translog 时，phase1 和 phase2 之间的写操作与 phase2 重放操作之间的时序错误和冲突，通过写流程中进行异常处理，对比版本号来过滤掉过期操作。

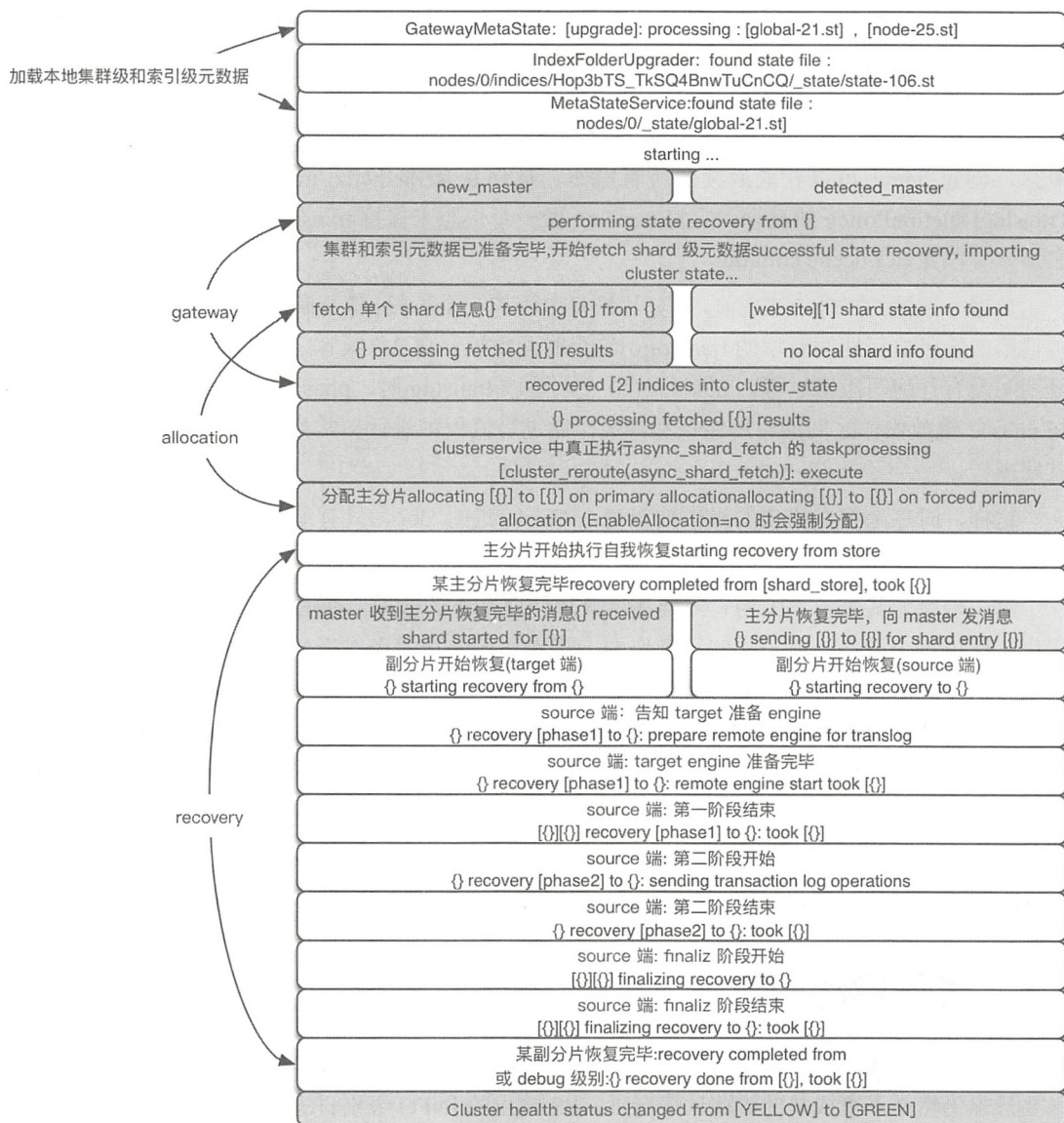
这样，时序上存在错误的操作被忽略，对于特定的 doc，只有最新一次操作生效，保证了主副分片一致。

第一阶段尤其漫长，因为它需要从主分片拉取全量的数据。在 ES 6.x 中，对第一阶段再次优化：标记每个操作。在正常的写操作中，每次写入成功的操作都分配一个序号，通过对比序号就可以计算出差异范围，在实现方式上，添加了 global checkpoint 和 local checkpoint，主分片负责维护 global checkpoint，代表所有分片都已写入这个序号的位置，local checkpoint 代表当前分片已写入成功的最新位置，恢复时通过对比两个序列号，计算出缺失的数据范围，然后通过 translog 重放这部分数据，同时 translog 会为此保留更长的时间。

因此，有两个机会可以跳过副分片恢复的 phase1：基于 SequenceNumber，从主分片节点的 translog 恢复数据；主副两分片有相同的 syncid 且 doc 数相同，可以跳过 phase1。

3.5 集群启动日志

日志是分布式系统中排查问题的重要手段，虽然 ES 提供了很多便于排查问题的接口，但重要日志仍然是不可或缺的。默认情况下，ES 输出的 INFO 级别日志较少，许多重要模块的关键环节是 DEBUG 或 TRACE 级别的，下图列出了集群启动过程相关的重要日志，部分调整到了 INFO 级别。



3.6 小结

当一个索引的主分片分配成功后, 到此分片的写操作就是允许的。当一个索引所有的主分片都分配成功后, 该索引变为 Yellow。当全部索引的主分片都分配成功后, 整个集群变为 Yellow。

当一个索引全部分片分配成功后，该索引变为 Green。当全部索引的索引分片分配成功后，整个集群变为 Green。

索引数据恢复是最漫长的过程。当 shard 总量达到十万级的时候，6.x 之前的版本集群从 Red 变为 Green 的时间可能需要小时级。ES 6.x 中的副本允许从本地 translog 恢复是一次重大的改进，避免了从主分片所在节点拉取全量数据，为恢复过程节约了大量时间。

4 chapter

第 4 章 节点的启动和关闭

本章分析单个节点的启动和关闭流程。看看进程是如何解析配置、检查环境、初始化内部模块的，以及在节点被“kill”的时候是如何处理的。

4.1 启动流程做了什么

总体来说，节点启动流程的任务是做下面几类工作：

- 解析配置，包括配置文件和命令行参数。
- 检查外部环境和内部环境，例如，JVM 版本、操作系统内核参数等。
- 初始化内部资源，创建内部模块，初始化探测器。
- 启动各个子模块和 keepalive 线程。

4.2 启动流程分析

4.2.1 启动脚本

当我们通过启动脚本 `bin/elasticsearch` 启动 ES 时，脚本通过 `exec` 加载 Java 程序。代码如下：


```

exec \      #执行命令
"$JAVA" \  #Java 程序路径
$ES_JAVA_OPTS \ #JVM 选项
-Des.path.home="$ES_HOME" \ #设置 path.home 路径
-Des.path.conf="$ES_PATH_CONF" \ #设置 path.conf 路径
-cp "$ES_CLASSPATH" \ #设置 java classpath
org.elasticsearch.bootstrap.Elasticsearch \ #指定 main 函数所在类
"$@" #传递给 main 函数命令行参数

```

ES_JAVA_OPTS 变量保存了 JVM 参数,其内容来自对 config/jvm.options 配置文件的解析。

如果执行启动脚本时添加了 -d 参数:

```
bin/elasticsearch -d
```

则启动脚本会在 exec 中添加 &&。&& 的作用是关闭标准输入,即进程中的 0 号 fd。& 的作用是让进程在后台运行。

4.2.2 解析命令行参数和配置文件

目前支持的命令行参数有下面几种,默认启动时都不使用,如下表所示。

参 数	含 义
-E	设定某项配置。例如, 设置集群名称: -E "cluster.name=my_cluster", 一般通过配置文件来设置, 而不是在命令行设置
-V, --version	打印版本号信息
-d, --daemonize	后台启动
-h, --help	打印帮助信息
-p, --pidfile	启动时在指定路径创建一个 pid 文件, 其中保存了当前进程的 pid, 之后可以通过查看这个 pid 文件来关闭进程
-q, --quiet	关闭控制台的标准输出和标准错误输出
-s, --silent	终端输出最少信息 (默认为 normal)
-v, --verbose	终端输出详细信息

实际工程应用中建议在启动参数中添加 -d 和 -p, 例如:

```
bin/elasticsearch -d -p es.pid
```

此处解析的配置文件有下面两个，jvm.options 是在启动脚本中解析的。

```
elasticsearch.yml #主要配置文件
log4j2.properties #日志配置文件
```

4.2.3 加载安全配置

什么是安全配置？本质上是配置信息，既然是配置信息，一般是写到配置文件中的。ES 的几个配置文件在之前的章节提到过。此处的“安全配置”是为了解决有些敏感的信息不适合放到配置文件中的，因为配置文件是明文保存的，虽然文件系统有基于用户权限的保护，但这仍然不够。因此 ES 把这些敏感配置信息加密，单独放到一个文件中：config/elasticsearch.keystore。然后提供一些命令来查看、添加和删除配置。

哪种配置信息适合放到安全配置文件中？例如，X-Pack 中的 security 相关配置，LDAP 的 base_dn 等信息（相当于登录服务器的用户名和密码）。

4.2.4 检查内部环境

内部环境指 ES 软件包本身的完整性和正确性。包括：

- 检查 Lucene 版本，ES 各版本对使用的 Lucene 版本是有要求的，在这里检查 Lucene 版本以防止有人替换不兼容的 jar 包。
- 检测 jar 冲突（JarHell），发现冲突则退出进程。

4.2.5 检测外部环境

ES 中的“节点”在实现时被封装为 Node 模块。在 Node 类中调用其他内部组件，同时对外提供启动和关闭方法，对外部环境的检测就是在 Node.start()中进行的。

外部环境指运行时的 JVM、操作系统相关参数，这些在 ES 中称为“Bootstrap Check”。在早期的 ES 版本中，ES 检测到一些不合理的配置会记录到日志中继续运行。但是有时候用户会错过这些日志。为了避免后期才发现问题，ES 在启动阶段对那些很重要的参数做检查，一些影响性能的配置会被标记为错误，让用户足够重视这些参数。

所有这些检查被单独封装在 BootstrapChecks 类中。目前有下面这些检测项。

1. 堆大小检查

如果 JVM 初始堆大小（Xms）与最大堆大小（Xmx）的值不同，则使用期间 JVM 堆大小

调整时可能会出现停顿。因此应该设置为相同值。

如果开启了 `bootstrap.memory_lock`, 则 JVM 将在启动时锁定堆的初始大小。如果初始堆大小与最大堆大小不同, 那么在堆大小发生变化后, 可能无法保证所有 JVM 堆都锁定在内存中。

要通过本项检查, 就必须配置堆大小。

2. 文件描述符检查

UNIX 架构的系统中, “文件” 可以是普通的物理文件, 也可以是虚拟文件, 网络套接字也是文件描述符。ES 进程需要非常多的文件描述符。例如, 每个分片有很多段, 每个段都有很多文件。同时包括许多与其他节点的网络连接等。

要通过此项检查, 就需要调整系统的默认配置, 在 Linux 下, 执行 `ulimit -n 65536` (只对当前终端生效), 或者在 `/etc/security/limits.conf` 文件中配置 “* - nofile 65536” (所有用户永久生效)。Ubuntu 下 `limits.conf` 默认被忽略, 需要开启 `pam_limits.so` 模块。

由于 Ubuntu 版本更新比较快, 而生产环境不适合频繁更新, 因此我们推荐使用 CentOS 作为服务器操作系统。

3. 内存锁定检查

ES 允许进程只使用物理内存, 避免使用交换分区。实际上, 我们建议生产环境中直接禁用操作系统的交换分区。现在已经不是因为内存不足而需要交换到硬盘上的时代, 对于服务器来说, 当内存真的用完时, 交换到硬盘上会引起更多问题。

开启 `bootstrap.memory_lock` 选项来让 ES 锁定内存, 在开启本项检查, 而锁定失败的情况下, 本项检查执行失败。

4. 最大线程数检查

ES 将请求分解为多个阶段执行, 每个阶段使用不同的线程池来执行。因此 ES 进程需要创建很多线程, 本项检查就是确保 ES 进程有创建足够多线程的权限。本项检查只对 Linux 系统进行。你需要调节进程可以创建的最大线程数, 这个值至少是 2048。

要通过这项检查, 可以修改 `/etc/security/limits.conf` 文件的 `nproc` 来完成配置。

5. 最大虚拟内存检查

Lucene 使用 `mmap` 来映射部分索引到进程地址空间, 最大虚拟内存检查确保 ES 进程拥有足够多的地址空间, 这项检查只对 Linux 执行。

要通过这项检查, 可以修改 `/etc/security/limits.conf` 文件, 设置 `as` 为 `unlimited`。

6. 最大文件大小检查

段文件和事务日志文件存储在本地磁盘中, 它们可能会非常大, 在有最大文件大小限制的

操作系统中，可能会导致写入失败。建议将最大文件的大小设置为无限。

要通过这项检查，可以修改/etc/security/limits.conf 文件，修改 fsize 为 unlimited。

7. 虚拟内存区域最大数量检查

ES 进程需要创建很多内存映射区，本项检查是要确保内核允许创建至少 262144 个内存映射区。该检查只对 Linux 执行。

要通过这项检查，可以执行下面的命令（临时生效，重启后失效）：

```
sysctl -w vm.max_map_count=262144
```

或者在/etc/sysctl.conf 文件中添加一行 vm.max_map_count=262144，然后执行下面的命令（立即，且永久生效）

```
sysctl -p
```

8. JVM Client 模式检查

OpenJDK 提供了两种 JVM 的运行模式：client JVM 模式与 server JVM 模式。client JVM 调优了启动时间和内存消耗，server JVM 提供了更高的性能。要想通过此检查，需要以 server 的方式来启动 ES，这也是默认的。

9. 串行收集检查

串行收集器（serial collector）适合单逻辑 CPU 的机器或非常小的堆，不适合 ES。使用串行收集器对 ES 有非常大的负面影响。本项检查就是确保没有使用串行收集器。ES 默认使用 CMS 收集器。

10. 系统调用过滤器检查

根据不同的操作系统，ES 安装各种不同的系统调用过滤器（在 Linux 下使用 seccomp）。这些过滤器可以阻止一些攻击行为。

作为一个服务端进程，当由于某些系统漏洞被攻击者取得进程的权限时，攻击者可以使用启动当前进程的用户权限执行一些操作。首先，以普通用户权限启动进程可以降低安全风险。其次，把服务本身不需要的系统调用通过过滤器关闭，当进程被攻击者取得权限时，进一步的权限提升等行为会增加攻击难度（例如，创建子进程执行其他程序，获得一个 shell 等）。这样被攻击的损失仅限于当前进程，而不是整个操作系统及其他数据。

要通过此项检查，可能需要解决过滤器安装期间遇到的错误，或者通过下面的设置来关闭系统调用过滤器：


```
bootstrap.system_call_filter: false
```

11. OnError 与 OnOutOfMemoryError 检查

如果 JVM 遇到致命错误（OnError）或 OutOfMemoryError（OnOutOfMemoryError），那么 JVM 选项 OnError 和 OnOutOfMemoryError 可以执行任意命令。

但是，默认情况下，ES 的系统调用过滤器是启用的（seccomp），fork 会被阻止。因此，使用 OnError 或 OnOutOfMemoryError 和系统调用过滤器不兼容。

若要通过此项检查，则不要启用 OnError 或 OnOutOfMemoryError，而是升级到 Java 8u92 并使用 ExitOnOutOfMemoryError。

12. Early-access 检查

OpenJDK 为即将发布的版本提供了 early-access 快照，这些发行版不适合生产环境。若要通过此项检查，则需要让 ES 运行在 JVM 的稳定版。

13. G1GC 检查

JDK 8 的早期版本有些问题，会导致索引损坏，JDK 8u40 之前的版本都会受影响。本项检查验证是否是早期的 HotSpot JVM 版本。

4.2.6 启动内部模块

环境检查完毕，开始启动各子模块。子模块在 Node 类中创建，启动它们时调用各自的 start() 方法，例如：

```
discovery.start();
clusterService.start();
nodeConnectionsService.start();
```

子模块的 start 方法基本就是初始化内部数据、创建线程池、启动线程池等操作。

4.2.7 启动 keepalive 线程

调用 keepAliveThread.start() 方法启动 keepalive 线程，线程本身不做具体的工作。主线程执行完启动流程后会退出，keepalive 线程是唯一的用户线程，作用是保持进程运行。在 Java 程序中，至少要有有一个用户线程。当用户线程数为零时退出进程。

4.3 节点关闭流程

现在我们探讨一下单个节点的关闭流程。设想当我们为 ES 集群更新配置、升级版本时，需要通过“kill”ES 进程来关闭节点。但是 kill 操作是否安全？如果此时节点有正在执行的读写操作会有什么影响？如果节点是 Master 该如何处理？关闭流程是怎么实现的？kill 节点都会带来哪些风险？

答案是：ES 进程会捕获 SIGTERM 信号（kill 命令默认信号）进行处理，调用各模块的 stop 方法，让它们有机会停止服务，安全退出。

进程重启期间，如果主节点被关闭，则集群会重新选主，在这期间，集群有一个短暂的无主状态。如果集群中的主节点是单独部署的，则新主当选后，可以跳过 gateway 和 recovery 流程，否则新主需要重新分配旧主所持有的分片：提升其他副本为主分片，以及分配新的副分片。

如果数据节点被关闭，则读写请求的 TCP 连接也会因此关闭，对客户端来说写操作执行失败。但写流程已经到达 Engine 环节的会正常写完，只是客户端无法感知结果。此时客户端重试，如果使用自动生成 ID，则数据内容会重复。

综合来说，滚动升级产生的影响是中断当前写请求，以及主节点重启可能引起的分片分配过程。提升新的主分片一般都比较快，因此对集群的写入可用性影响不大。

当索引部分主分片未分配时，使用自动生成 ID 的情况下，如果持续写入，则客户端对失败重试可能会成功（请求到达已分配成功的主分片），但是会在不同的分片之间产生数据倾斜，倾斜程度视期间数量而定。

4.4 关闭流程分析

在节点启动过程中，Bootstrap#setup 方法中添加了 shutdown hook，当进程收到系统 SIGTERM（kill 命令默认信号）或 SIGINT 信号时，调用 Node#close 方法，执行节点关闭流程。

每个模块的 Service 中都实现了 doStop 和 doClose，用于处理这个模块的正常关闭流程。节点总的关闭流程位于 Node#close，在 close 方法的实现中，先调用一遍各个模块的 doStop，然后再次遍历各个模块执行 doClose。主要实现代码如下：

```
if (lifecycle.started()) {  
    stop(); //调用各模块的 doStop 方法  
}
```

```

List<Closeable> toClose = new ArrayList<>();
//在 toClose 中添加所有需要关闭的 Service, 以 nodeService 为例
toClose.add(nodeService);
.....

//调用各模块 doClose 方法
IOUtils.close(toClose);

```

各模块的关闭有一定的顺序关系, 以 doStop 为例, 按下表所示的顺序调用各模块 doStop 方法。

服 务	简 介
ResourceWatcherService	通用资源监视服务
HttpServerTransport	HTTP 传输服务, 提供 REST 接口服务
SnapshotsService	快照服务
SnapshotShardsService	负责启动和停止 shard 级快照
IndicesClusterStateService	收到集群状态信息后, 处理其中索引相关操作
Discovery	集群拓扑管理
RoutingService	处理 reroute (节点之间迁移 shard)
ClusterService	集群管理服务, 主要处理集群任务, 发布集群状态
NodeConnectionsService	节点连接管理服务
MonitorService	提供进程级、系统级、文件系统和 JVM 的监控服务
GatewayService	负责集群元数据持久化与恢复
SearchService	处理搜索请求
TransportService	底层传输服务
plugins	当前的所有插件
IndicesService	负责创建、删除索引等索引操作

综合来看, 关闭顺序大致如下:

- 关闭快照和 HTTPServer, 不再响应用户 REST 请求。
- 关闭集群拓扑管理, 不再响应 ping 请求。
- 关闭网络模块, 让节点离线。
- 执行各个插件的关闭流程。
- 关闭 IndicesService。

最后才关闭 `IndicesService`，是因为这期间需要等待释放的资源最多，时间最长。

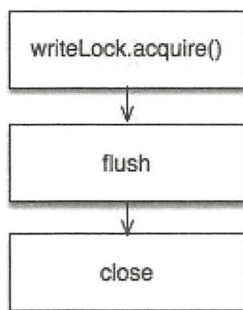
4.5 分片读写过程中执行关闭

下面分别对读和写执行过程中关闭节点进行分析。

写入过程中关闭：线程在写入数据时，会对 `Engine` 加写锁。`IndicesService` 的 `doStop` 方法对本节点上全部索引并行执行 `removeIndex`，当执行到 `Engine` 的 `flushAndClose`（先 `flush` 然后关闭 `Engine`），也会对 `Engine` 加写锁。由于写入操作已经加了写锁，此时写锁会等待，直到写入执行完毕。因此数据写入过程不会被中断。但是由于网络模块被关闭，客户端的连接会被断开。客户端应当作为失败处理，虽然 `ES` 服务端的写流程还在继续。

读取过程中关闭：线程在读取数据时，会对 `Engine` 加读锁。`flushAndClose` 时的写锁会等待读取过程执行完毕。但是由于连接被关闭，无法发送给客户端，导致客户端读失败。

下图展示了 `Engine` 的 `flushAndClose` 过程。



节点关闭过程中，`IndicesService` 的 `doStop` 对 `Engine` 设置了超时，如果 `flushAndClose` 一直等待，则 `CountDownLatch.await` 默认 1 天才会继续后面的流程。

4.6 主节点被关闭

主节点被关闭时，没有想象中的特殊处理，节点正常执行关闭流程，当 `TransportService` 模块被关闭后，集群重新选举新 `Master`。因此，滚动重启期间会有一段时间处于无主状态。

4.7 小结

(1) 总体来说，节点启动流程做的就是初始化和检查工作，各个子模块启动后异步地工作，

加载本地数据，或者选主、加入集群等，在后面的章节中单独介绍。

（2）节点在关闭时有机会处理未写完的数据，但是写完后可能来不及通知客户端。包括线程池中尚未执行的任务，在一定的超时时间内都有机会执行完。

集群健康从 Red 变为 Green 的时间主要消耗在维护主副分片的一致性上。我们也可以选择在集群健康为 Yellow 时就允许客户端写入，但是会牺牲一些数据安全性。



5 chapter

第 5 章 选主流程

Discovery 模块负责发现集群中的节点，以及选择主节点。ES 支持多种不同 Discovery 类型选择，内置的实现称为 Zen Discovery，其他的包括公有云平台亚马逊的 EC2、谷歌的 GCE 等。本章讨论内置的 Zen Discovery 实现。Zen Discovery 封装了节点发现 (Ping)、选主等实现过程，现在我们先讨论选主流程，在后面的章节中整体性介绍 Discovery 模块。

5.1 设计思想

所有分布式系统都需要以某种方式处理一致性问题。一般情况下，可以将策略分为两组：试图避免不一致及定义发生不一致之后如何协调它们。后者在适用场景下非常强大，但对数据模型有比较严格的限制。因此这里研究前者，以及如何应对网络故障。

5.2 为什么使用主从模式

除主从 (Leader/Follower) 模式外，另一种选择是分布式哈希表 (DHT)，可以支持每小时数千个节点的离开和加入，其可以在不了解底层网络拓扑的异构网络中工作，查询响应时间大约为 4 到 10 跳 (中转次数)，例如，Cassandra 就使用这种方案。但是在相对稳定的对等网络中，主从模式会更好。

ES 的典型场景中的另一个简化是集群中没有那么多节点。通常，节点的数量远远小于单个节点能够维护的连接数，并且网络环境不必经常处理节点的加入和离开。这就是为什么主从模



式更适合 ES。

5.3 选举算法

在主节点选举算法的选择上，基本原则是不重复造轮子。最好实现一个众所周知的算法，这样的好处是其中的优点和缺陷是已知的。ES 的选举算法的选择上主要考虑下面两种。

1. Bully 算法

Leader 选举的基本算法之一。它假定所有节点都有一个唯一的 ID，使用该 ID 对节点进行排序。任何时候的当前 Leader 都是参与集群的最高 ID 节点。该算法的优点是易于实现。但是，当拥有最大 ID 的节点处于不稳定状态的场景下会有问题。例如，Master 负载过重而假死，集群拥有第二大 ID 的节点被选为新主，这时原来的 Master 恢复，再次被选为新主，然后又假死……

ES 通过推迟选举，直到当前的 Master 失效来解决上述问题，只要当前主节点不挂掉，就不重新选主。但是容易产生脑裂（双主），为此，再通过“法定得票人数过半”解决脑裂问题。

2. Paxos 算法

Paxos 非常强大，尤其在什么时机，以及如何进行选举方面的灵活性比简单的 Bully 算法有很大的优势，因为在现实生活中，存在比网络连接异常更多的故障模式。但 Paxos 实现起来非常复杂。

5.4 相关配置

与选主过程相关的重要配置有下列几个，并非全部配置。

`discovery.zen.minimum_master_nodes`：最小主节点数，这是防止脑裂、防止数据丢失的极其重要的参数。这个参数的实际作用早已超越了其表面的含义。除了在选主时用于决定“多数”，还用于多处重要的判断，至少包含以下时机：

- **触发选主** 进入选主的流程之前，参选的节点数需要达到法定人数。
- **决定 Master** 选出临时的 Master 之后，这个临时 Master 需要判断加入它的节点达到法定人数，才确认选主成功。
- **gateway 选举元信息** 向有 Master 资格的节点发起请求，获取元数据，获取的响应数量必须达到法定人数，也就是参与元信息选举的节点数。
- **Master 发布集群状态** 发布成功数量为多数。



为了避免脑裂，它的值应该是半数以上（quorum）：

```
(master_eligible_nodes / 2) + 1
```

例如，如果有 3 个具备 Master 资格的节点，则这个值至少应该设置为 $(3/2) + 1 = 2$ 。

该参数可以动态设置：

```
PUT /_cluster/settings
{
  "persistent" : {
    "discovery.zen.minimum_master_nodes" : 2
  }
}
```

`discovery.zen.ping.unicast.hosts`：集群的种子节点列表，构建集群时本节点会尝试连接这个节点列表，那么列表中的主机会看到整个集群中都有哪些主机。可以配置为部分或全部集群节点。可以像下面这样指定：

```
discovery.zen.ping.unicast.hosts:
- 192.168.1.10:9300
- 192.168.1.11
- seeds.mydomain.com
```

默认使用 9300 端口，如果需要更改端口号，则可以在 IP 后手工指定端口。也可以设置一个域名，让该域名解析到多个 IP 地址，ES 会尝试连接这个 IP 列表中的全部地址。

`discovery.zen.ping.unicast.hosts.resolve_timeout`：DNS 解析超时时间，默认为 5 秒。

`discovery.zen.join_timeout`：节点加入现有集群时的超时时间，默认为 `ping_timeout` 的 20 倍。

`discovery.zen.join_retry_attempts join_timeout`：超时之后的重试次数，默认为 3 次。

`discovery.zen.join_retry_delay join_timeout`：超时之后，重试前的延迟时间，默认为 100 毫秒。

`discovery.zen.master_election.ignore_non_master_pings`：设置为 `true` 时，选主阶段将忽略来自不具备 Master 资格节点（`node.master: false`）的 ping 请求，默认为 `false`。

`discovery.zen.fd.ping_interval`：故障检测间隔周期，默认为 1 秒。



`discovery.zen.fd.ping_timeout`: 故障检测请求超时时间, 默认为 30 秒。

`discovery.zen.fd.ping_retries`: 故障检测超时后的重试次数, 默认为 3 次。

5.5 流程概述

ZenDiscovery 的选主过程如下:

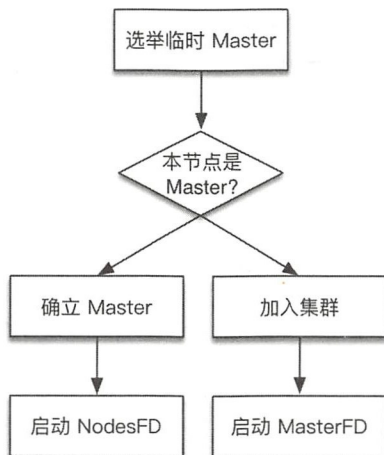
- 每个节点计算最小的已知节点 ID, 该节点为临时 Master。向该节点发送领导投票。
- 如果一个节点收到足够多的票数, 并且该节点也为自己投票, 那么它将扮演领导者的角色, 开始发布集群状态。

所有节点都会参与选举, 并参与投票, 但是, 只有有资格成为 Master 的节点 (`node.master` 为 `true`) 的投票才有效。

获得多少选票可以赢得选举胜利, 就是所谓的法定人数。在 ES 中, 法定大小是一个可配置的参数。配置项: `discovery.zen.minimum_master_nodes`。为了避免脑裂, 最小值应该是有 Master 资格的节点数 $n/2+1$ 。

5.6 流程分析

整体流程可以概括为: 选举临时 Master, 如果本节点当选, 则等待确立 Master, 如果其他节点当选, 则尝试加入集群, 然后启动节点失效探测器。具体如下图所示。



执行本流程的线程池: `generic`。

下面我们具体分析每个步骤的实现。



5.6.1 选举临时 Master

选举过程的实现位于 `ZenDiscovery#findMaster`。该函数查找当前集群的活跃 Master，或者从候选者中选择新的 Master。如果选主成功，则返回选定的 Master，否则返回空。

为什么是临时 Master？因为还需要等待下一个步骤，该节点的得票数足够时，才确立为真正的 Master。

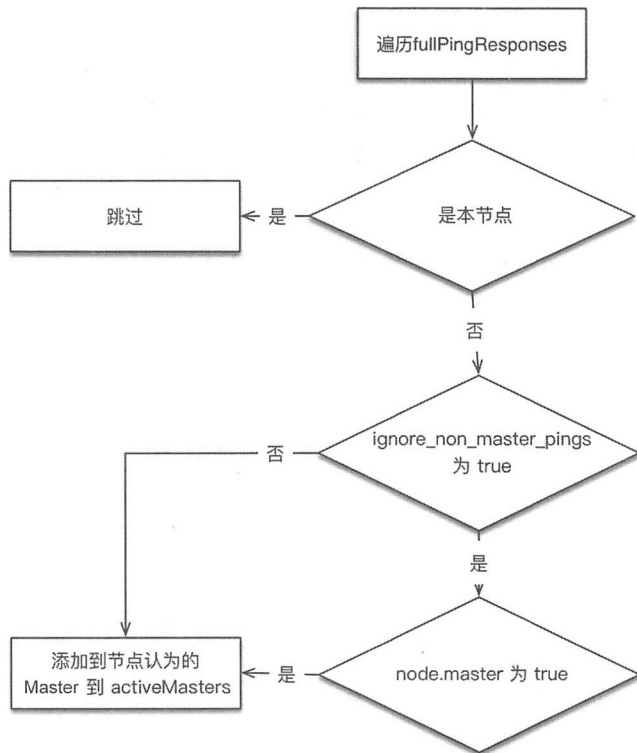
临时 Master 的选举过程如下：

(1) “ping” 所有节点，获取节点列表 `fullPingResponses`，ping 结果不包含本节点，把本节点单独添加到 `fullPingResponses` 中。

(2) 构建两个列表。

`activeMasters` 列表：存储集群当前活跃 Master 列表。遍历第一步获取的所有节点，将每个节点所认为的当前 Master 节点加入 `activeMasters` 列表中（不包括本节点）。在遍历过程中，如果配置了 `discovery.zen.master_election.ignore_non_master_pings` 为 `true`（默认为 `false`），而节点又不具备 Master 资格，则跳过该节点。

具体流程如下图所示。



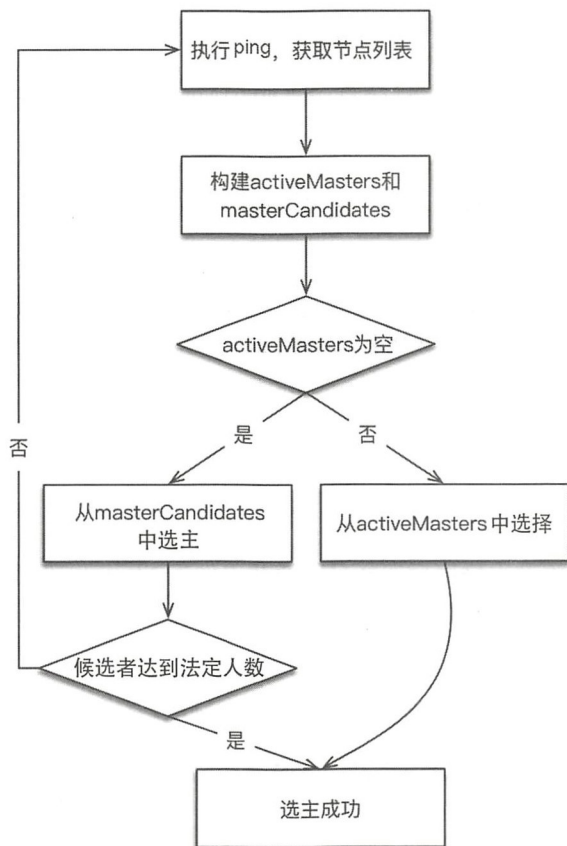


这个过程是将集群当前已存在的 Master 加入 activeMasters 列表，正常情况下只有一个。如果集群已存在 Master，则每个节点都记录了当前 Master 是哪个，考虑到异常情况下，可能各个节点看到的当前 Master 不同。在构建 activeMasters 列表过程中，如果节点不具备 Master 资格，则可以通过 ignore_non_master_pings 选项忽略它认为的那个 Master。

masterCandidates 列表：存储 master 候选者列表。遍历第一步获取列表，去掉不具备 Master 资格的节点，添加到这个列表中。

(3) 如果 activeMasters 为空，则从 masterCandidates 中选举，结果可能选举成功，也可能选举失败。如果不为空，则从 activeMasters 中选择最合适的作为 Master。

整体流程如下图所示。



从 masterCandidates 中选主

与选主的具体细节实现封装在 ElectMasterService 类中，例如，判断候选者是否足够，选择具体的节点作为 Master 等。



从 masterCandidates 中选主时，首先需要判断当前候选者人数是否达到法定人数，否则选主失败。

```
public boolean hasEnoughCandidates(Collection<MasterCandidate> candidates) {
    //候选者为空，返回失败
    if (candidates.isEmpty()) {
        return false;
    }

    //默认值为-1，确保单节点的集群可以正常选主
    if (minimumMasterNodes < 1) {
        return true;
    }
    return candidates.size() >= minimumMasterNodes;
}
```

当候选者人数达到法定人数后，从候选者中选一个出来做 Master：

```
public MasterCandidate electMaster(Collection<MasterCandidate> candidates) {
    List<MasterCandidate> sortedCandidates = new ArrayList<>(candidates);
    //通过自定义的比较函数对候选者节点从小到大排序
    sortedCandidates.sort(MasterCandidate::compare);
    //返回最新的作为 Master
    return sortedCandidates.get(0);
}
```

可以看出这里只是将节点排序后选择最小的节点作为 Master。但是排序时使用自定义的比较函数 MasterCandidate::compare，早期的版本中只是对节点 ID 进行排序，现在会优先把集群状态版本号高的节点放在前面。

使用默认比较函数的情况下，sort 结果为从小到大排序。参考 Long 类型的比较函数的实现：

```
public static int compare(long x, long y) {
    return (x < y) ? -1 : ((x == y) ? 0 : 1);
}
```

自定义比较函数的实现：

```
public static int compare(MasterCandidate c1, MasterCandidate c2) {
```



```
//先比较集群状态版本，注意此处 c2 在前，c1 在后
int ret = Long.compare(c2.clusterStateVersion, c1.clusterStateVersion);
//如果版本号相同，则比较节点 ID
if (ret == 0) {
    ret = compareNodes(c1.getNode(), c2.getNode());
}
return ret;
}
```

节点比较函数 `compareNodes` 的实现：对于排序效果来说，如果传入的两个节点中，有一个节点具备 Master 资格，而另一个不具备，则把有 Master 资格的节点排在前面。如果都不具备 Master 资格，或者都具备 Master 资格，则比较节点 ID。

但是，`masterCandidates` 列表中的节点都是具备 Master 资格的。`compareNodes` 比较函数的两个 if 判断是因为在别的函数调用中会存在节点列表中可能存在不具备 Master 资格节点的情况。因此此处只会比较节点 ID。

```
private static int compareNodes(DiscoveryNode o1, DiscoveryNode o2) {
    //两个 if 处理两节点中一个具备 Master 资格而另一个不具备的情况
    if (o1.isMasterNode() && !o2.isMasterNode()) {
        return -1;
    }
    if (!o1.isMasterNode() && o2.isMasterNode()) {
        return 1;
    }
    //通过节点 ID 排序
    return o1.getId().compareTo(o2.getId());
}
```

从 activeMasters 列表中选择

列表存储着集群当前存在活跃的 Master，从这些已知的 Master 节点中选择一个作为选举结果。选择过程非常简单，取列表中的最小值，比较函数仍然通过 `compareNodes` 实现，`activeMasters` 列表中的节点理论情况下都是具备 Master 资格的。

```
public DiscoveryNode tieBreakActiveMasters(Collection<DiscoveryNode>
activeMasters) {
    return activeMasters.stream().min(ElectMasterService::compareNodes).get();
}
```




5.6.2 投票与得票的实现

在 ES 中，发送投票就是发送加入集群（JoinRequest）请求。得票就是申请加入该节点的请求的数量。

收集投票，进行统计的实现在 ZenDiscovery#handleJoinRequest 方法中，收到的连接被存储到 ElectionContext#joinRequestAccumulator 中。当节点检查收到的投票是否足够时，就是检查加入它的连接数是否足够，其中会去掉没有 Master 资格节点的投票。

```
public synchronized int getPendingMasterJoinsCount() {
    int pendingMasterJoins = 0;
    //遍历当前收到的 join 请求
    for (DiscoveryNode node : joinRequestAccumulator.keySet()) {
        //过滤不具备 master 资格的节点
        if (node.isMasterNode()) {
            pendingMasterJoins++;
        }
    }
    return pendingMasterJoins;
}
```

5.6.3 确立 Master 或加入集群

选举出的临时 Master 有两种情况：该临时 Master 是本节点或非本节点。为此单独处理。现在准备向其发送投票。

如果临时 Master 是本节点：

- (1) 等待足够多的具备 Master 资格的节点加入本节点（投票达到法定人数），以完成选举。
- (2) 超时（默认为 30 秒，可配置）后还没有满足数量的 join 请求，则选举失败，需要进行新一轮选举。
- (3) 成功后发布新的 clusterState。

如果其他节点被选为 Master：

- (1) 不再接受其他节点的 join 请求。
- (2) 向 Master 发送加入请求，并等待回复。超时时间默认为 1 分钟（可配置），如果遇到异常，则默认重试 3 次（可配置）。这个步骤在 joinElectedMaster 方法中实现。
- (3) 最终当选的 Master 会先发布集群状态，才确认客户的 join 请求，因此，joinElectedMaster

返回代表收到了 join 请求的确认，并且已经收到了集群状态。本步骤检查收到的集群状态中的 Master 节点如果为空，或者当选的 Master 不是之前选择的节点，则重新选举。

5.7 节点失效检测

到此为止，选主流程已执行完毕，Master 身份已确认，非 Master 节点已加入集群。

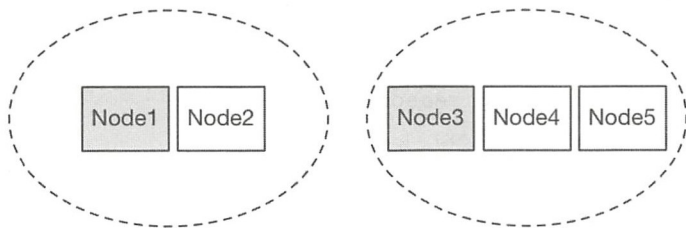
节点失效检测会监控节点是否离线，然后处理其中的异常。失效检测是选主流程之后不可或缺的步骤，不执行失效检测可能会产生脑裂（双主或多主）。在此我们需要启动两种失效探测器：

- 在 Master 节点，启动 NodesFaultDetection，简称 NodesFD。定期探测加入集群的节点是否活跃。
- 在非 Master 节点启动 MasterFaultDetection，简称 MasterFD。定期探测 Master 节点是否活跃。

NodesFaultDetection 和 MasterFaultDetection 都是通过定期（默认为 1 秒）发送的 ping 请求探测节点是否正常的，当失败达到一定次数（默认为 3 次），或者收到来自底层连接模块的节点离线通知时，开始处理节点离开事件。

5.7.1 NodesFaultDetection 事件处理

检查一下当前集群总节点数是否达到法定节点数（过半），如果不足，则会放弃 Master 身份，重新加入集群。为什么要这么做？设想下面的场景，如下图所示。



假设有 5 台机器组成的集群产生网络分区，2 台组成一组，另外 3 台组成一组，产生分区前，原 Master 为 Node1。此时 3 台一组的节点会重新选举并成功选取 Node3 作为 Master，不会产生双主？

NodesFaultDetection 就是为了避免上述场景下产生双主。

对应事件处理主要实现如下：在 ZenDiscovery#handleNodeFailure 中执行 NodeRemoval-ClusterStateTaskExecutor#execute。

```

    public ClusterTasksResult<Task> execute(final ClusterState currentState,
final List<Task> tasks) throws Exception {
        //判断剩余节点是否达到法定人数
        if (electMasterService.hasEnoughMasterNodes(remainingNodesClusterState.
nodes()) == false) {
            final int masterNodes = electMasterService.countMasterNodes
(remainingNodesClusterState.nodes());
            rejoin.accept(LoggerMessageFormat.format("not enough master nodes
(has [{}], but needed [{}])",
                masterNodes, electMasterService.minimumMasterNodes()));
            return resultBuilder.build(currentState);
        } else {
            return resultBuilder.build(allocationService.deassociateDeadNodes
(remainingNodesClusterState, true, describeTasks(tasks)));
        }
    }

```

主节点在探测到节点离线的事件处理中, 如果发现当前集群节点数量不足法定人数, 则放弃 Master 身份, 从而避免产生双主。

5.7.2 MasterFaultDetection 事件处理

探测 Master 离线的处理很简单, 重新加入集群。本质上就是该节点重新执行一遍选主的流程。

对应事件处理主要实现如下: ZenDiscovery#handleMasterGone。

```

    private void handleMasterGone(final DiscoveryNode masterNode, final
Throwable cause, final String reason) {
        synchronized (stateMutex) {
            if (localNodeMaster() == false && masterNode.equals(committedState.
get().nodes().getMasterNode())) {
                pendingStatesQueue.failAllStatesAndClear(new ElasticsearchException
("master left [{}]", reason));
                //重新加入集群
                rejoin("master left (reason = " + reason + ")");
            }
        }
    }
}

```

5.8 小结

选主流程在集群中启动，从无主状态到产生新主时执行，同时集群在正常运行过程中，Master 探测到节点离开，非 Master 节点探测到 Master 离开时都会执行。



第 6 章

数据模型

6.1 PacificA 算法

ES 的数据副本模型基于主从模式（或称主备模式，HDFS 和 Cassandra 为对等模式），在实现过程中参考了微软的 PacificA 算法（借鉴了其中部分思想，并非完全按照这个模型实现）。我们先看一下 PacificA 算法的几个特点：

- 设计了一个通用的、抽象的框架，而不是具体的、特定的算法。模型的正确性容易验证。
- 配置管理和数据副本分离，Paxos 负责管理配置，数据副本策略采取主从模式。
- 将错误检测和配置更新放在数据副本的交互里实现，去中心化。

该算法涉及的几个术语如下。

- **Replica Group**: 一个互为副本的数据集合称为副本组。其中只有一个副本是主数据（Primary），其他为从数据（Secondary）。
- **Configuration**: 配置信息中描述了一个副本组都有哪些副本，Primary 是谁，以及它们位于哪个节点。
- **Configuration Version**: 配置信息的版本号，每次发生变更时递增。
- **Serial Number**: 代表每个写操作的顺序，每次写操作时递增，简称 SN。每个主副本维护自己的递增 SN。

- **Prepared List:** 写操作的准备序列。存储来自外部请求的列表，将请求按照 SN 排序，向列表中插入的序列号必须大于列表中最大的 SN。每个副本上有自己的 Prepared List。
- **Committed List:** 写操作的提交序列。

设计前提与假设：

- 节点可以失效，对消息延迟的上限不做假设。
- 消息可以丢失、乱序，但不能被篡改，即不存在拜占庭问题。
- 网络分区可以发生，系统时钟可以不同步，但漂移是有限度的。

整个系统框架主要由两部分组成：存储管理和配置管理。

- **存储管理：**负责数据的读取和更新，使用多副本方式保证数据的可靠性和可用性；
- **配置管理：**对配置信息进行管理，维护所有配置信息的一致性。

6.1.1 数据副本策略

分片副本使用主从模式。多个副本中存在一个主副本 **Primary** 和多个从副本 **Secondary**。所有的数据写入操作都进入主副本，当主副本出现故障无法访问时，系统从其他从副本中选择合适的副本作为新的主副本。

数据写入的流程如下：

(1) 写请求进入主副本节点，节点为该操作分配 SN，使用该 SN 创建 **UpdateRequest** 结构。然后将该 **UpdateRequest** 插入自己的 **prepare list**。

(2) 主副本节点将携带 SN 的 **UpdateRequest** 发往从副本节点，从节点收到后同样插入 **prepare list**，完成后给主副本节点回复一个 **ACK**。

(3) 一旦主副本节点收到所有从副本节点的响应，确定该数据已经被正确写入所有的从副本节点，此时认为可以提交了，将此 **UpdateRequest** 放入 **committed list**，**committed list** 向前移动。

(4) 主副本节点回复客户端更新成功完成。对每一个 **Prepare** 消息，主副本节点向从副本节点发送一个 **commit** 通知，告诉它们自己的 **committed point** 位置，从副本节点收到通知后根据指示移动 **committed point** 到相同的位置。

因为主副本只有在所有从副本将请求添加进 **prepared list** 之后才可以移动 **committed point** 的方式将该请求插入 **committed list** 中，因此主副本的 **committed list** 是任何一个从副本的 **prepared list** 的前缀（或者称为子集）。例如，从副本 **prepared list** 中 SN 为 1、2、3、4；主副本 **committed point** 中 SN 一定不会大于 4，如 1、2、3。

同时，因为一个从副本只有在主副本将一个请求添加进 **committed list** 后才会把同样的请求

添加进 committed list 中, 因此一个从副本上的 committed list 是主副本上 committed list 的前缀, 此不变式称为 Commit Invariant。

令 P 为主副本, R 为从副本, 以下不变式成立: $\text{committed_R} \subseteq \text{committed_P} \subseteq \text{prepared_R}$ 。

6.1.2 配置管理

全局的配置管理器负责管理所有副本组的配置。节点可以向管理器提出添加/移除副本的请求, 每次请求都需要附带当前配置版本号, 只有这个版本号和管理器记录的版本号一致才会被执行, 如果请求成功, 则这个新配置会被赋予新的版本号。

6.1.3 错误检测

分布式系统经常存在网络分区、节点离线等异常。全局的配置管理器维护权威配置信息, 但其他各节点上的配置信息不一定同步, 我们必须处理旧的主副本和新的主副本同时存在的情况——旧的主副本可能没有意识到重新分配了一个新的主副本, 从而违反了强一致性。Pacifica 使用了租约 (lease) 机制来解决这个问题。

主副本定期向其他从副本获取租约。这个过程中可能产生两种情况:

- 如果主副本节点在一定时间内 (lease period) 未收到从副本节点的租约回复, 则主副本节点认为从副本节点异常, 向配置管理器汇报, 将该异常从副本从副本组中移除, 同时, 它也将自己降级, 不再作为主副本节点。
- 如果从副本节点在一定时间内 (grace period) 未收到主副本节点的租约请求, 则认为主副本异常, 向配置管理器汇报, 将主副本从副本组中移除, 同时将自己提升为新的主。如果存在多个从副本, 则哪个从副本先执行成功, 哪个从副本就被提升为新主。

假设没有时钟漂移, 只要 $\text{grace period} \geq \text{lease period}$, 则租约机制就可以保证主副本会比任意从副本先感知到租约失效。同时任何一个从副本只有在它租约失效时才会争取去当新的主副本, 因此保证了新主副本产生之前, 旧的主分片已降级, 不会产生两个主副本。

其他系统也经常将租约机制作为故障检测手段, 如 GFS、Bigtable。

Pacifica 算法的这些概念对应 ES 中:

- Master 负责维护索引元信息, 类似配置管理器维护配置信息。
- 集群状态中的 routing_table 存储了所有索引、索引有哪些 shard、各自的主分片, 以及位于哪个节点等信息, 类似副本组。
- SequenceNumber 和 Checkpoint 类似 Pacifica 算法中的 Serial Number 和 Committed Point。

6.2 ES 的数据副本模型

ES 中的每个索引都会被拆分为多个分片，并且每个分片都有多个副本。这些副本称为 `replication group`（副本组，与 PacificA 中的副本组概念一致），并且在删除或添加文档的时候，各个副本必须同步。否则，从不同副本中读取的数据会不一致。我们把保持分片副本之间的同步，以及从中读取的过程称为数据副本模型（`data replication model`）。

ES 的数据副本模型基于主备模式（`primary-backup model`），主分片是所有索引操作的入口，它负责验证索引操作是否有效。一旦主分片接受一个索引操作，主分片的副分片也会接受该操作。

下面讨论数据副本模型在写操作和读操作时如何交互。

6.2.1 基本写入模型

每个索引操作首先会使用 `routing` 参数解析到副本组，通常基于文档 ID。一旦确定副本组，就会内部转发该操作到分片组的主分片中。主分片负责验证操作和转发它到其他副分片。ES 维护一个可以接收该操作的分片的副本列表。这个列表叫作同步副本列表（`in-sync copies`），并由 Master 节点维护。正如它的名字，这个“好”分片副本列表中的分片，都会保证已成功处理所有的索引和删除操作，并给用户返回 `ACK`。主分片负责维护不变性（各个副本保持一致），因此必须复制这些操作到这个列表中的每个副本。

写入过程遵循以下基本流程：

（1）请求到达协调节点，协调节点先验证操作，如果有错就拒绝该操作。然后根据当前集群状态，请求被路由到主分片所在节点。

（2）该操作在主分片上本地执行，例如，索引、更新或删除文档。这也会验证字段的内容，如果未通过就拒绝操作（例如，字段串的长度超出 Lucene 定义的长度）。

（3）操作成功执行后，转发该操作到当前 `in-sync` 副本组的所有副分片。如果有多个副分片，则会并行转发。

（4）一旦所有的副分片成功执行操作并回复主分片，主分片会把请求执行成功的信息返回给协调节点，协调节点返回给客户端。

6.2.2 写故障处理

写入期间可能会发生很多错误——硬盘损坏、节点离线，或者某些配置错误，这些错误都可能导致无法在副分片上执行某个操作，虽然这比较少见，但是主分片必须汇报这些错误信息。

对于主分片自身错误的情况，它所在的节点会发送一个消息到 Master 节点。这个索引操作会等待（默认为最多一分钟）Master 节点提升一个副分片为主分片。这个操作会被转发给新的主分片。注意，Master 同样会监控节点的健康，并且可能会主动降级主分片。这通常发生在主分片所在的节点离线的时候。

在主分片上执行的操作成功后，该主分片必须处理在副分片上潜在发生的错误。错误发生的原因可能是在副分片上执行操作时发生的错误，也可能是因为网络阻塞，导致主分片无法转发操作到副分片，或者副分片无法返回结果给主分片。这些错误都会导致相同的结果：in-sync replica set 中的一个分片丢失一个即将要向用户确认的操作。为了避免出现不一致，主分片会发送一条消息到 Master 节点，要求它把有问题的分片从 in-sync replica set 中移除。一旦 Master 确认移除了该分片，主分片就会确认这次操作。注意，Master 也会指导另一个节点建立一个新的分片副本，以便把系统恢复成健康状态。

在转发请求到副分片时，主分片会使用副分片来验证它是否仍是一个活跃的主分片。如果主分片因为网络原因（或很长时间的 GC）被隔离，则在它意识到被降级之前可能会继续处理传入的索引操作。来自陈旧的主分片的操作将会被副分片拒绝。当它接收来自副分片的拒绝其请求的响应时，它将会访问一下主节点，然后就会知道自己已被替换。最后将操作路由到新的主分片。

如果没有副分片呢？

出现这种场景可能是因为索引配置或所有副分片都发生故障。在这种情况下，主分片处理的操作没有经过任何外部验证，可能会导致问题。另一方面，主分片节点将副分片失效的消息告知主节点，主节点知道主分片是唯一可用的副本。因此我们确保主节点不会提升任何其他分片副本（过时的）为主分片，并且索引到主分片上的任何操作都不会丢失。当然，由于只运行单个数据副本，当物理硬件出问题可能会丢失数据。可以使用 `wait_for_active_shards` 缓解此类问题。

6.2.3 基本读取模型

通过 ID 读取是非常轻量级的操作，而一个巨大的复杂的聚合查询请求需要消耗大量 CPU 和内存资源。主从模式的一个好处是保证所有的分片副本都是一致的（正在执行的操作例外）。因此，单个 in-sync 中的某个副本也可以提供服务。当一个读请求被协调节点接收，这个节点负

责转发它到其他涉及相关分片的节点，并整理响应结果发送给客户端。接收用户请求的这个节点称为协调节点。

基本流程如下：

(1) 把读请求转发到相关分片。注意，因为大多数搜索都会发送到一个或多个索引，通常需要从多个分片中读取，每个分片都保存这些数据的一部分。

(2) 从副本组中选择一个相关分片的活跃副本。它可以是主分片或副分片。默认情况下，ES 会简单地循环遍历这些分片。

(3) 发送分片级的读请求到被选中的副本。

(4) 合并结果并给客户端返回响应。注意，针对通过 ID 查找的 get 请求，会跳过这个步骤，因为只有一个相关的分片。

6.2.4 读故障处理

当分片不能响应一个读请求时，协调节点会从副本组中选择另一个副本，将请求转发给它。没有可用的分片副本会导致重复的错误。在某些情况下，例如，`_search`，ES 倾向于尽早响应，即使只有部分结果，也不等待问题被解决（可以在响应结果的 `_shards` 字段中检查本次结果是完整的还是部分的）。

6.2.5 引申的含义

基本流程决定了 ES 系统在读和写时的表现。此外，由于读写可以同时执行，所以这两个基本流程互相有些影响。这有一些固定的含义。

高效读取

在正常操作下，读操作在相关副本组中只执行一次。只有在出错的时候，才会在同一个分片的不同副本中执行多次。

在写操作返回应答之前读取

主分片首先在本地进行索引，然后转发请求，由于主分片已经写成功，因此在并行的读请求中，有可能在写请求返回成功之前就可以读取更新的内容。

默认两副本

在只有 2 个副本的情况下，该模型也是可以容错的。这与 quorum-based 的系统相反，其容错的最小副本为 3。

6.2.6 系统异常

在出现故障时，可能产生下面的情况。

只有单个分片可能降低索引速度

因为每次操作时主分片会等待所有在 `in-sync` 列表中的副本，所以单个缓慢的副本可能降低整个副本组的写速度。当然，单个缓慢的分片也会降低读取速度。

脏读

从一个被隔离的主分片进行读取，可能读取没有经过确认的写操作。这是因为只有主分片向副分片转发请求，或者向主节点发送请求的时候才会被隔离，此时数据已经在主分片写成功，可以被读取到。ES 通过定期（默认为 1 秒）“ping”主节点来降低这种风险，如果没有已知的主节点，则拒绝索引操作。

6.3 Allocation IDs

ES 从 5.x 版本开始引入 Allocation IDs 的概念，用于主分片选举策略。每个分片有自己唯一的 Allocation ID，同时集群元信息中有一个列表，记录了哪些分片拥有最新数据。

ES 通过在集群中保留多个数据副本的方式提供故障转移功能，当出现网络分区或节点挂掉时，更改操作可能无法在所有副本上完成，此时我们希望把写失败的副本标记出来。

ES 的数据副本模型会假定其中一个数据副本为权威副本，称之为**主分片**。所有的索引操作写主分片，完成后，主分片所在节点会负责把更改转发到活跃的备份副本，称之为**副分片**。如果当前主分片临时或永久地变为不可用状态，则另一个分片副本将被提升为主分片。因为主分片是权威的数据副本，因此在这个模型中，只把含有最新数据的分片作为主分片是至关重要的。如果将一个旧数据的分片作为主分片，则它将作为最终副本，从而导致这个副本之后的数据将会丢失。下面我们介绍如何追踪到那个可以安全地被选为主分片的副本，也称之为**同步(in-sync)分片副本**。

6.3.1 安全地分配主分片

分片分配就是决定哪个节点应该存储一个活跃分片的过程。分片决策过程在主节点完成，并记录在集群状态中，该数据结构还包含了其他元数据，如索引设置及映射。分配决策包含两部分：哪个分片应该分配到哪个节点，以及哪个分片作为主分片，哪些作为副分片。主节点广播集群状态到集群的所有节点。这样每个节点都有了集群状态，它们就可以实现对请求的智能

路由。因为每个节点都知道主副分片分配到了哪里。

每个节点都会通过检查集群状态来判断某个分片是否可用。如果一个分片被指定为主分片，则这个节点只需要加载本地分片副本，使之可以用于搜索即可。如果一个分片被分配为副分片，则节点首先需要从主分片所在节点复制差异数据。当集群中可用副分片不足时（在索引设置中指定：`index.number_of_replicas`），主节点也可以将副分片分配到不含任何此分片副本的节点，从而指示这些节点创建主分片的完整副本。

在创建新索引时，主节点在选择哪个节点作为主分片方面有很大的灵活性，会将集群均衡和其他约束（如分配感知及过滤器）考虑在内。分配已存在的分片副本为主分片是比较少见的情况，例如，集群完全重启（`full restart`），所有分片都是未分配状态，或者短时间内所有活跃副本都变为不可用。在这种情况下，主节点询问所有节点，找到磁盘中存在的分片副本，根据找到的副本，决定是否将其中一个作为主分片。为了确保安全，主节点必须确保被选为主分片的副本含有最新数据。为此，ES 使用 `Allocation IDs` 的概念，这是区分不同分片的唯一标识（`UUIDS`）。

`Allocation IDs` 由主节点在分片分配时指定，并由数据节点存储在磁盘中，紧邻实际的数据分片。主节点负责追踪包含最新数据副本的子集。这些副本集合称为同步分片标识（`in-sync allocation IDs`），存储于集群状态中。集群状态存在于集群的主节点和所有数据节点。对集群状态的更改由 `zen discovery` 模块实现一致性支持。它确保集群中有共同的理解，即哪些分片副本被认为是同步的（`in-sync`），隐式地将那些不在同步集合中的分片副本标记为陈旧（`stale`）。

也就是说，`Allocation IDs` 存储在 `shard` 级元信息中，每个 `shard` 都有自己唯一的 `Allocation ID`，同时集群级元信息中记录了一个被认为是最新 `shard` 的 `Allocation ID` 集合，这个集合称为 **`in-sync allocation IDs`**。

当分配主分片时，主节点检查磁盘中存储的 `Allocation ID` 是否会在集群状态的 `in-sync allocations IDs` 集合中出现，只有在这个集合中找到了，此分片才有可能被选为主分片。如果活跃副本中的主分片挂了，则 `in-sync` 集合中的活跃分片会被提升为主分片，确保集群的写入可用性不变。

6.3.2 将分配标记为陈旧

在 `Elasticsearch` 中，数据副本和元信息副本使用不同的副本策略，元信息的改变需要在集群层面达成一致，而数据副本使用简单的主备方法。系统的这两个层面可以使数据副本更简单、更快。只有在特殊情况下才需要与集群一致（`consensus`）层交互。处理写请求过程中，当网络产生分区、节点故障，或者部分节点未启动，主分片本地执行完写操作，转发到副分片时，转发操作可能在一个或多个副分片上没能执行成功，这意味着主分片中含有一些没有传播到所有

分片的数据，如果这些副分片仍然被认为是同步的，那么即使它们遗漏了一些变化，它们也可能稍后被选为主分片，结果丢失数据。

解决这种问题有两种方法：

- (1) 让写请求失败，已经写的做回滚处理。
- (2) 确保差异的（divergent）分片不再被视为同步。

ES 在这种情况下选择了写入可用性：主分片所在节点命令主节点将差异分片的 Allocation IDs 从同步集合（in-sync set）中删除。然后，主分片所在节点等待主节点删除成功的确认消息，这个确认消息意味着集群一致层（consensus layer）已成功更新，之后才向客户端确认写请求。这样确保只有包含了所有已确认写入的分片副本才会被主节点选为主分片。

6.2.3 一个例子

我们跟随一个简单的例子（来自官网）来说明上述情况，在一个小型集群中包含一个主节点和两个数据节点。为了保持例子简单，我们创建只有 1 个主分片和 1 个副分片的索引。最初一个数据节点拥有主分片，另一个数据节点拥有副分片。我们使用 cluster state API 来查阅集群状态中的 in-sync 分片信息，并使用 “filter_path” query 参数过滤出感兴趣的结果：

```
GET /_cluster/state?filter_path=metadata.indices.my_index.in_sync_
allocations.*,routing_table.indices.my_index.*
```

产生以下集群状态的摘要：

```
{
  "metadata": {
    "indices": {
      "my_index": {
        "in_sync_allocations": {
          "0": [
            "HNeGpt5aS3W9it3a7tJusg",
            "wP-Z5fuGSM-HbADjMNpSIQ"
          ]
        }
      }
    }
  },
  "routing_table": {
```

```

"indices": {
  "my_index": {
    "shards": {
      "0": [
        {
          "primary": true,
          "state": "STARTED",
          "allocation_id": { "id": "HNeGpt5aS3W9it3a7tJusg" },
          "node": "CX-rFmoPQF21tgt3MYGSQA",
          ...
        },
        {
          "primary": false,
          "state": "STARTED",
          "allocation_id": { "id": "wP-Z5fuGSM-HbADjMNpSIQ" },
          "node": "AzYoyzzSSwG6v_ypdRXYkw",
          ...
        }
      ]
    }
  }
}

```

集群状态显示出主分片和副分片都已启动，主分片分配在数据节点“CX-rFmo”，副分片分配在数据节点“AzYoyz”。它们都有唯一的 `allocation_id`，同时出现在 `in_sync_allocations` 集合中。

让我们看看当关闭主分片所在节点时会发生什么。由于这并不改变分片上的数据，所以两个分片副本应该保持同步。在没有主分片的情况下，副分片也应该被提升为主分片，这些都会反映在集群状态中：

```

{
  "metadata": {
    "indices": {
      "my_index": {
        "in_sync_allocations": {

```

```
"0": [
    "HNeGpt5aS3W9it3a7tJusg",
    "wP-Z5fuGSM-HbADjMNpSIQ"
]
}
}
},
"routing_table": {
    "indices": {
        "my_index": {
            "shards": {
                "0": [
                    {
                        "primary": true,
                        "state": "STARTED",
                        "allocation_id": { "id": "wP-Z5fuGSM-HbADjMNpSIQ" },
                        "node": "AzYoyzzSSwG6v_ypdRXYkw",
                        ...
                    },
                    {
                        "primary": false,
                        "state": "UNASSIGNED",
                        "node": null,
                        "unassigned_info": {
                            "details": "node_left[CX-rFmoPQF2ltgt3MYGSQA]",
                            ...
                        }
                    }
                ]
            }
        }
    }
}
```

由于只有一个数据节点，所以副分片停留在未分配状态。如果我们再次启动第二个节点，则副分片将自动分配在这个节点上。为了使这个场景更有趣，我们不启动第二个节点，相反，

我们索引一个文档到新提升的主分片中。由于分片副本现在是差异的 (diverging)，不活跃的那个分片副本变为陈旧的，因此它的 ID 被主节点从 in-sync 集合中删除：

```
{
  "metadata": {
    "indices": {
      "my_index": {
        "in_sync_allocations": {
          "0": [
            "wP-Z5fuGSM-HbADjMNpSIQ"
          ]
        }
      }
    }
  },
  "routing_table": {
    ... // 与上一步的相同
  }
}
```

现在只剩下一个同步的分片副本，让我们看看如果该副本变为不可用，那么系统将如何处理。为此，我们关闭当前唯一的数据节点，然后启动前一个拥有陈旧分片副本的数据节点，之后 cluster health API 显示 cluster health 为 Red，集群状态显示主分片尚未分配：

```
{
  "metadata": {
    "indices": {
      "my_index": {
        "in_sync_allocations": {
          "0": [
            "wP-Z5fuGSM-HbADjMNpSIQ"
          ]
        }
      }
    }
  },
  "routing_table": {
    "indices": {
```



```

    "my_index": {
      "shards": {
        "0": [
          {
            "primary": true,
            "state": "UNASSIGNED",
            "recovery_source": { "type": "EXISTING_STORE" },
            "unassigned_info": {
              "allocation_status": "no_valid_shard_copy",
              "at": "2017-01-26T09:20:24.054Z",
              "details": "node_left[AzYoyzzSSwG6v_ypdRXYkw]"
            },
            ...
          },
          {
            "primary": false,
            "state": "UNASSIGNED",
            "recovery_source": { "type": "PEER" },
            "unassigned_info": {
              "allocation_status": "no_attempt",
              "at": "2017-01-26T09:14:47.689Z",
              "details": "node_left[CX-rFmoPQF21tgt3MYGSQA]"
            },
            ...
          }
        ]
      }
    }
  }
}

```

让我们再看看 cluster allocation explain API，这是一个调试分配问题的好工具。运行不带参数的 explain 命令将提供系统找到的第一个未分配分片的说明：

```
GET /_cluster/allocation/explain
```

explain API 告诉我们为什么主分片处于未分配状态，同时还提供了基于每个节点上的更详

细的分配信息。在这个例子中，主节点在集群当前可用节点中无法找到同步的（in-sync）分片副本。

```
{
  "index" : "my_index",
  "shard" : 0,
  "primary" : true,
  "current_state" : "unassigned",
  "unassigned_info" : {
    "reason" : "NODE_LEFT",
    "at" : "2017-01-26T09:20:24.054Z",
    "last_allocation_status" : "no_valid_shard_copy"
  },
  "can_allocate" : "no_valid_shard_copy",
  "allocate_explanation" : "cannot allocate because all found copies of the
shard are either stale or corrupt",
  "node_allocation_decisions" : [
    {
      "node_id" : "CX-rFmoPQF21tgt3MYGSQA",
      "node_name" : "CX-rFmo",
      "transport_address" : "127.0.0.1:9301",
      "node_decision" : "no",
      "store" : {
        "in_sync" : false,
        "allocation_id" : "HNeGpt5aS3W9it3a7tJusg"
      }
    }
  ]
}
```

该 API 还显示在节点“CY-rFmo”上可用的分片副本是陈旧的（store.in_sync = false）。启动拥有 in-sync 分片副本的那个节点将使集群的状态重新变为 Green。如果那个节点永远都不回来了呢？

6.3.4 不会丢失全部

发生严重灾难时，集群中可能会出现只有陈旧副本可用的情况。ES 不会把这些分片自动分

配为主分片，集群将持续保持 Red 状态。如果所有 in-sync 副本都消失了，则集群仍有可能使用陈旧副本进行恢复，但这需要管理员手工干预。

正如我们在前面的例子中看到的，理解分片问题的第一步是使用 `cluster allocation explain` API。它显示节点是否具有分片的副本，以及相应的副本是否处于同步集合中。在文件系统损坏的情况下，它也显示了访问磁盘信息时抛出的异常。在前面的例子中，`allocation explain` 的输出显示在集群中的节点“CX-rFmo”上找到了现有的分片副本，但该副本未包含最新的数据（in-sync: false）

`reroute` API 提供了一个子命令 `allocate_stale_primary`，用于将一个陈旧的分片分配为主分片。使用此命令意味着丢失给定分片副本中缺少的数据。如果同步分片副本只是暂时不可用，则使用此命令意味着会丢失同步分片副本中最近更新的数据。应该把它看作使集群至少运行一些数据的最后一种措施。在所有分片副本都不存在的情况下，还可以强制 ES 使用空分片副本分配主分片，这意味着丢失与该分片相关联的所有先前数据。不言而喻，`allocate_empty_primary` 命令只能用于最糟糕的情况，其含义很好理解。

6.4 Sequence IDs

ES 从 6.0 版本开始引入了 Sequence IDs 概念，使用唯一的 ID 来标记每个写操作。通过这个 ID 我们有了索引操作的总排序。

写操作先到达主分片，主分片写完后转发到副分片，在转发到副分片之前，增加一个计数器，为每个操作分配一个序列号是很简单的。但是，由于节点离线随时可能发生，例如，网络分区等，主分片可能被其他副分片取代，仅仅由主分片分配一个序列号无法保证全局唯一性和单调性。因此，我们把当前主分片做一个标记，放到每个操作中，这就是 Primary Terms。这样，来自旧的主分片的迟到的操作就可以被检测到然后拒绝（虽然 Allocation IDs 可以让主分片分配在拥有最新数据的分片上，但仍然可能存在某些情况下主分片上的数据并非最新，例如，手工分配主分片到有旧数据的副本）。

6.4.1 Primary Terms 和 Sequence Numbers

第一步是能够区分新旧两种主分片，我们必须找到一种方法来识别是来自较旧的主分片操作还是来自较新的主分片的操作。最重要的是，整个集群需要达成一致。为此，我们添加了 Primary Terms。它由主节点分配，当一个主分片被提升时，Primary Terms 递增。然后持久化到集群状态中，从而表示集群主分片所处的一个版本。有了 Primary Terms，操作历史中的任何冲突都可以通过查看操作的 Primary Terms 来解决。新的 Terms 优先于旧 Terms，拒绝过时的操作，

避免混乱的情况。

一旦我们有了 Primary Terms 的保护, 就可以添加一个简单的计数器, 给每个操作分配一个 Sequence Numbers (序列号)。Sequence Numbers 使我们能够理解发生在主分片节点上的索引操作的特定顺序, 接下来讨论 Sequence Numbers 带来的各种好处。可以在 Response 中看到分配的 Sequence Numbers 和 Primary Terms:

```
curl -H 'Content-Type: application/json' -XPOST http://127.0.0.1:9200/
foo/doc?pretty -d '{ "bar": "baz" }'
{
  "_index" : "foo",
  "_type" : "doc",
  "_id" : "MlDBm10BditXXu4kjj5E",
  "_version" : 1,
  "result" : "created",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 19,
  "_primary_term" : 1
}
```

我们再次整理一下这两个概念:

- Primary Terms 由主节点分配给每个主分片, 每次主分片发生变化时递增。这和 Raft 中的 term, 以及 Zab 中 Viewstamped Replication 的 view-number 概念很相似。
- Sequence Numbers 标记发生在某个分片上的写操作。由主分片分配, 只对写操作分配。假设索引 website 有 2 个主分片和 1 个副分片, 当分片 website[0] 的序列号增加到 5 时, 它的主分片离线, 副分片被提升为新的主分片, 对于后续写操作, 序列号从 6 开始递增。分片 website[1] 有自己独立的序列号计数器。

主分片每次向副分片转发写请求时, 会带上这两个值。

为了实现将操作排序, 当我们比较两个操作 o1 和 o2 时, 如果 $o1 < o2$, 那么意味着:

```
s1.seq# < s2.seq#
```

或者

```
(s1.seq# == s2.seq# and s1.term < s2.term)
```

“等于”和以上“大于”以类似的方式定义。

6.4.2 本地及全局检查点

有了 Primary Terms 和 Sequence Numbers，我们就有了在理论上能够检测出分片之间差异，并在主分片失效时，重新对齐它们的工具。旧主分片就可以恢复为与拥有更高 Primary Terms 值的新主分片一致：从旧主分片中删除新主分片操作历史中不存在的操作，并将缺少的操作索引到旧主分片。

遗憾的是，当同时为每秒成百上千的事件做索引时，比较数百万个操作的历史是不切实际的。存储成本非常昂贵，直接进行比较的计算工作量太大。为了解决这个问题，ES 维护了一个名为“全局检查点”（global checkpoint）的安全标记。

全局检查点是所有活跃分片历史都已对齐的序列号，换句话说，所有低于全局检查点的操作都保证已被所有活跃的分片处理完毕。这意味着，当主分片失效时，我们只需要比较新主分片与其他副分片之间的最后一个全局检查点之后的操作即可。当旧主分片恢复时，我们使用它知道的全局检查点，与新主分片进行比较。这样，我们只有小部分操作需要比较，不用比较全部。

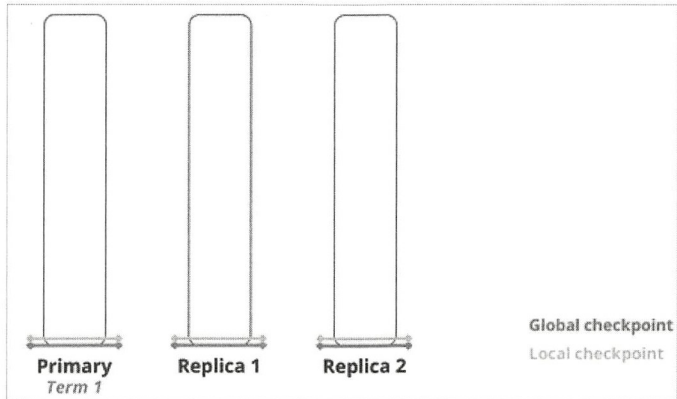
主分片负责推进全局检查点，它通过跟踪在副分片上完成的操作来实现。一旦它检测到所有副分片已经超出给定序列号，它将相应地更新全局检查点。副分片不会跟踪所有操作，而是维护一个类似全局检查点局部变量，称为本地检查点。

本地检查点也是一个序列号，所有序列号低于它的操作都已在该分片上处理（Lucene 和 translog 写成功，不一定刷盘）完毕。当副分片确认（ACK）一个写操作到主分片节点时，它们也会更新本地检查点。使用本地检查点，主分片节点能够更新全局检查点，然后在下一次索引操作时将其发送到所有分片副本。

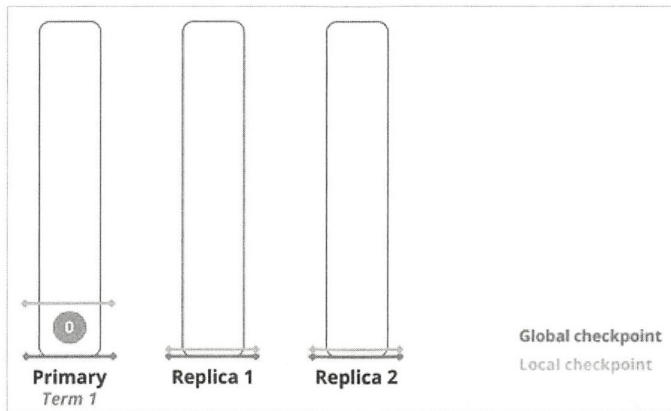
全局检测点和本地检查点在内存中维护，但也会保存在每个 Lucene 提交的元数据中。

下面演示在写入过程中，全局/本地检查点的更新情况（来自官网）。

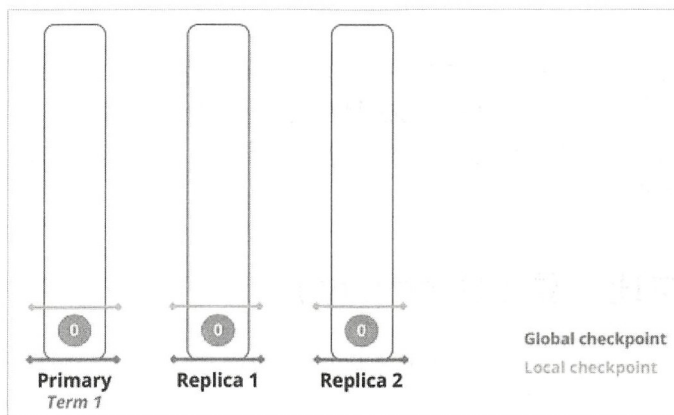
某索引有 1 个主分片、2 个副分片，初始状态没有数据，全局检查点和本地检查点都在 0 位置，如下图所示。



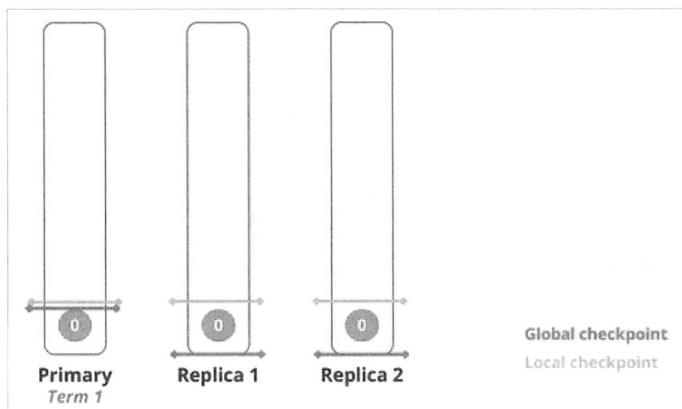
主分片写入一条数据成功后，本地检查点向前推进，如下图所示。



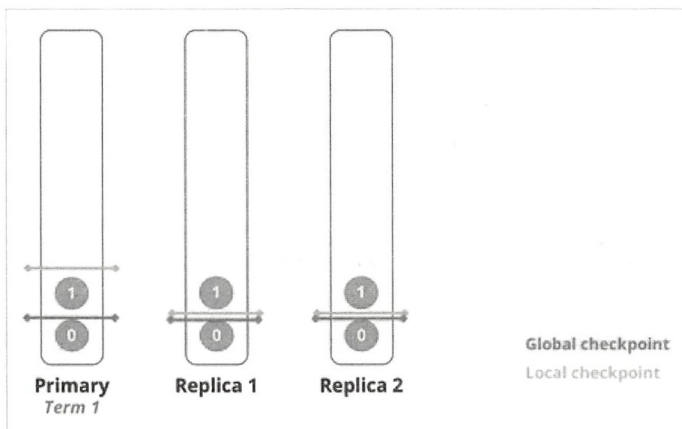
主分片将写请求转发到副分片，副分片本地处理成功后，将本地检查点向前推进，如下图所示。



主分片收到所有副分片都处理成功的消息，根据汇报的各副本上的本地检查点更新全局检查点，如下图所示。



在下次索引操作时，主分片节点将全局检查点发送到所有分片副本，如下图所示。



全局检查点还有另外一个很好的属性——它代表了已经保证存盘的操作边界（存储在所有活跃分片中）。如果主分片故障，则数据没有来得及复制到副分片，该区域（大于全局检测点的）可以包含可能需要回滚的操作。这是一个微妙且重要的属性，对于未来的更改 API 或跨数据中心复制功能来说至关重要。

6.4.3 用于快速恢复（Recovery）

当 ES 恢复一个分片时，需要保证恢复之后与主分片一致。对于冷数据来说，synced flush 可以快速验证副分片与主分片是否相同，但对于热数据来说，恢复过程需要从主分片复制整个

Lucene 分段，如果分段很大，则是非常耗时的操作。

现在我们使用副本所知道的最后一个全局检查点，重放来自主分片事务日志（translog）中的相关更改。也就是说，现在可以计算出待恢复分片与主分片数据的差异范围，因此避免复制整个分片。同时，我们多保留一些事务日志（默认为 512MB，12 小时），直到“太大”或“太老”。如果不能从事务日志恢复，则使用旧的恢复模式。

6.5 _version

每个文档都有一个版本号（_version），当文档被修改时版本号递增。ES 使用这个 _version 来确保变更以正确顺序执行。如果旧版本的文档在新版本之后到达，则它可以被简单地忽略。例如，索引 recovery 阶段就利用了这个特性。

版本号由主分片生成，在将请求转发给副本片时将携带此版本号。

版本号的另一个作用是实现乐观锁，如同其他数据库的乐观锁一样。我们在写请求中指定文档的版本号，如果文档的当前版本与请求中指定的版本号不同，则请求会失败。

例如（示例来自官网），先写入一条文档：

```
curl -XPUT 'localhost:9200/website/blog/1/_create?pretty' -H 'Content-Type: application/json' -d'
{
  "title": "My first blog entry",
  "text": "Just trying this out..."
}
```

获得的响应信息如下：

```
{
  "_index" : "website",
  "_type" : "blog",
  "_id" : "1",
  "_version" : 1,
  "result" : "created",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  }
}
```

```
    },  
    "_seq_no" : 0,  
    "_primary_term" : 5  
  }  
}
```

从响应体中可以看到，这个新创建的文档的版本号为 1。现在假设我们想编辑这个文档，当再次更新这个文档时，指定一个版本号：

```
curl -XPUT 'localhost:9200/website/blog/1?version=1&pretty' -H 'Content-Type: application/json' -d'  
{  
  "title": "My first blog entry",  
  "text": "Starting to get the hang of this..."  
}'
```

在这个更新请求中，只有版本号设置为 1，本次更新才能成功。否则 ES 会返回 409 Conflict 状态码。

7 chapter

第 7 章 写流程

本章分析 ES 写入单个和批量文档写请求的处理流程,仅限于 ES 内部实现,并不涉及 Lucene 内部处理。在 ES 中,写入单个文档的请求称为 Index 请求,批量写入的请求称为 Bulk 请求。写单个和多个文档使用相同的处理逻辑,请求被统一封装为 BulkRequest。

在分析写流程时,我们把流程按不同节点执行的操作进行划分。写请求的例子可以参考上一章。

7.1 文档操作的定义

在 ES 中,对文档的操作有下面几种类型:

```
enum OpType {  
    INDEX(0),  
    CREATE(1),  
    UPDATE(2),  
    DELETE(3);  
}
```

- INDEX: 向索引中“put”一个文档的操作称为“索引”一个文档。此处“索引”为动词。
- CREATE: put 请求可以通过 op_type 参数设置操作类型为 create,在这种操作下,如果文档已存在,则请求将失败。
- UPDATE: 默认情况下,“put”一个文档时,如果文档已存在,则更新它。

- DELETE：删除文档。

在 put API 中，通过 `op_type` 参数来指定操作类型。

7.2 可选参数

Index API 和 Bulk API 有一些可选参数，这些参数在请求的 URI 中指定，例如：

```
PUT my-index/my-type/my-id?pipeline=my_pipeline_id
{
  "foo": "bar"
}
```

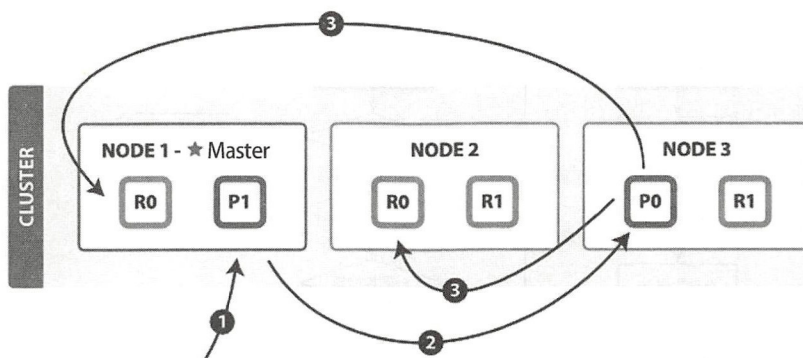
下面简单介绍各个参数的作用，这些参数在接下来的流程分析中都会遇到，如下表所示。

参 数	简 介
version	设置文档版本号。主要用于实现乐观锁
version_type	默认为 <code>internal</code> ，请求参数指定的版本号与存储的文档版本号相同则写入。其他可选值有 <code>external</code> 等类型，为 <code>external</code> 类型时，如果当前存储的文档版本号小于请求参数指定的版本号，则写入数据。 <code>version_type</code> 主要控制版本号的比较机制，用于对文档进行并发更新操作时同步数据
op_type	可设置为 <code>create</code> 。代表仅在文档不存在时才写入。如果文档已存在，则写请求将失败
routing	ES 默认使用文档 ID 进行路由，指定 <code>routing</code> 可使用 <code>routing</code> 值进行路由
wait_for_active_shards	用于控制写一致性，当指定数量的分片副本可用时才执行写入，否则重试直至超时。默认为 1，主分片可用即执行写入
refresh	写入完毕后执行 <code>refresh</code> ，使其对搜索可见
timeout	请求超时时间，默认为 1 分钟
pipeline	指定事先创建好的 <code>pipeline</code> 名称

7.3 Index/Bulk 基本流程

新建、索引（这里的索引是动词，指写入操作，将文档添加到 Lucene 的过程称为索引一个文档）和删除请求都是写操作。写操作必须先在主分片执行成功后才能复制到相关的副分片。

写单个文档的流程（图片来自官网）如下图所示。



以下是写单个文档所需的步骤：

(1) 客户端向 NODE1 发送写请求。

(2) NODE1 使用文档 ID 来确定文档属于分片 0，通过集群状态中的内容路由表信息获知分片 0 的主分片位于 NODE3，因此请求被转发到 NODE3 上。

(3) NODE3 上的主分片执行写操作。如果写入成功，则它将请求并行转发到 NODE1 和 NODE2 的副分片上，等待返回结果。当所有的副分片都报告成功，NODE3 将向协调节点报告成功，协调节点再向客户端报告成功。

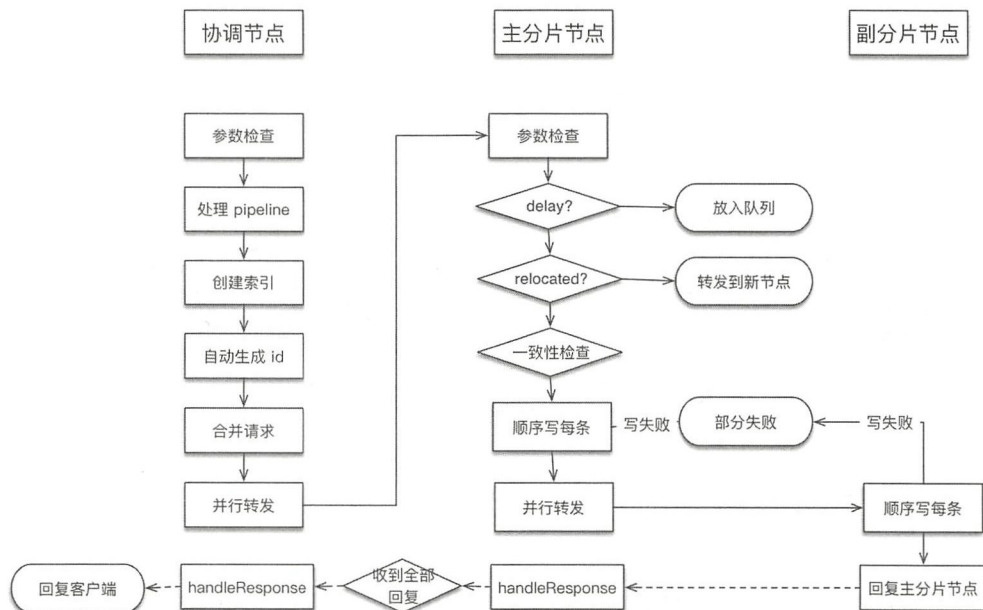
在客户端收到成功响应时，意味着写操作已经在主分片和所有副分片都执行完成。

写一致性的默认策略是 **quorum**，即多数的分片（其中分片副本可以是主分片或副分片）在写入操作时处于可用状态。

```
quorum = int( (primary + number_of_replicas) / 2 ) + 1
```

7.4 Index/Bulk 详细流程

以不同角色节点执行的任务整理流程如下图所示。



下面分别讨论各个节点上执行的流程。

7.4.1 协调节点流程

协调节点负责创建索引、转发请求到主分片节点、等待响应、回复客户端。

实现位于 `TransportBulkAction`。执行本流程的线程池：`http_server_worker`。

1. 参数检查

如同我们平常设计的任何一个对外服务的接口处理一样，收到用户请求后首先检测请求的合法性，把检查操作放在处理流程的第一步，有问题就直接拒绝，对异常请求的处理代价是最小的。

检查操作进行以下参数检查，如下表所示。

参 数	检 查
index	不可为空
type	不可为空
source	不可为空
contentType	不可为空
opType	当前操作类型如果是创建索引，则校验 <code>VersionType</code> 必须为 <code>internal</code> ，且 <code>Version</code> 不可为 <code>MATCH_DELETED</code>

续表

参 数	检 查
resolvedVersion	校验解析的 Version 是否合法
versionType	不可为 FORCE 类型，此类型已废弃
id	非空时，长度不可大于 512，以及为空时对 versionType 和 resolvedVersion 的检查

每项检查遇到异常都会拒绝当前请求。

2. 处理 pipeline 请求

数据预处理（ingest）工作通过定义 pipeline 和 processors 实现。pipeline 是一系列 processors 的定义，processors 按照声明的顺序执行。添加一个 pipeline 的简单例子如下：

```
PUT _ingest/pipeline/my-pipeline-id
{
  "description" : "describe pipeline",
  "processors" : [
    {
      "set" : {
        "field": "foo",
        "value": "bar"
      }
    }
  ]
}
```

my-pipeline-id 是自定义的 pipeline 名称，processors 中定义了一系列的处理器，本例中只有 set。

如果 Index 或 Bulk 请求中指定了 pipeline 参数，则先使用相应的 pipeline 进行处理。如果本节点不具备预处理资格，则将请求随机转发到其他具备预处理资格的节点。预处理节点资格的配置参考第 1 章中的节点角色。

3. 自动创建索引

如果配置为允许自动创建索引（默认允许），则计算请求中涉及的索引，可能有多个，其中有哪些索引是不存在的，然后创建它。如果部分索引创建失败，则涉及创建失败索引的请求被标记为失败。其他索引正常执行写流程。

创建索引请求被发送到 Master 节点，待收到全部创建请求的 Response（无论成功还是失败的）之后，才进入下一个流程。Master 节点什么时候返回 Response？在 Master 节点执行完创建

索引流程，将新的 clusterState 发布完毕才会返回。那什么才算发布完毕呢？默认情况下，Master 发布 clusterState 的 Request 收到半数以上的节点 Response，认为发布成功。负责写数据的节点会先执行一遍内容路由的过程以处理没有收到最新 clusterState 的情况。

简化的实现如下：

```
//遍历所有需要创建的索引
for (String index : autoCreateIndices) {
    //发送创建索引请求
    createIndex(index, bulkRequest.timeout(), new ActionListener
<CreateIndexResponse>() {
        //下面是 listener 的定义
        //收到执行成功响应
        public void onResponse(CreateIndexResponse result) {
            //将计数器递减，计数器的值为需要创建的索引数量
            if (counter.decrementAndGet() == 0) {
                //全部创建完毕时执行后面的流程，参数省略
                executeBulk(...);
            }
        }
        //收到失败的响应
        public void onFailure(Exception e) {
            //将创建失败索引对应的请求置空
            for (int i = 0; i < bulkRequest.requests.size(); i++) {
                if (request != null && setResponseFailureIfIndexMatches(...))
                    bulkRequest.requests.set(i, null);
            }
            if (counter.decrementAndGet() == 0) {
                executeBulk(...);
            }
        }
    });
}
```

4. 对请求的预先处理

这里不同于对数据的预处理，对请求的预先处理只是检查参数、自动生成 id、处理 routing 等。

由于上一步可能有创建索引操作，所以在此先获取最新集群状态信息。然后遍历所有请求，从集群状态中获取对应索引的元信息，检查 mapping、routing、id 等信息。如果 id 不存在，则生成一个 UUID 作为文档 id。

实现位于 `TransportBulkAction.BulkOperation#doRun`。

5. 检测集群状态

协调节点在开始处理时会先检测集群状态，若集群异常则取消写入。例如，Master 节点不存在，会阻塞等待 Master 节点直至超时。

```
final ClusterState clusterState = observer.setAndGetObservedState();
if (handleBlockExceptions(clusterState)) {
    return;
}
```

因此索引为 Red 时，如果 Master 节点存在，则数据可以写到正常 shard，Master 节点不存在，协调节点会阻塞等待或取消写入。

6. 内容路由，构建基于 shard 的请求

将用户的 `bulkRequest` 重新组织为基于 shard 的请求列表。例如，原始用户请求可能有 10 个写操作，如果这些文档的主分片都属于同一个，则写请求被合并为 1 个。所以这里本质上是合并请求的过程。此处尚未确定主分片节点。

基于 shard 的请求结构如下：

```
Map<ShardId, List<BulkItemRequest>> requestsByShard = new HashMap<>();
```

根据路由算法计算某文档属于哪个分片。遍历所有的用户请求，重新封装后添加到上述 map 结构。

`ShardId` 类的主要结构如下，shard 编号是从 0 开始递增的序号：

```
public class ShardId {
    //分片所属的索引
    private Index index;
    //从 0 开始的递增的序号
    private int shardId;
    private int hashCode;
}
```

7. 路由算法

路由算法就是根据 routing 和文档 id 计算目标 shardid 的过程。

一般情况下，路由计算方式为下面的公式：

$$\text{shard_num} = \text{hash}(_routing) \% \text{num_primary_shards}$$

默认情况下，_routing 值就是文档 id。

ES 使用随机 id 和 Hash 算法来确保文档均匀地分配给分片。当使用自定义 id 或 routing 时，id 或 routing 值可能不够随机，造成数据倾斜，部分分片过大。在这种情况下，可以使用 index.routing_partition_size 配置来减少倾斜的风险。routing_partition_size 越大，数据的分布越均匀。

在设置了 index.routing_partition_size 的情况下，计算公式为：

$$\text{shard_num} = (\text{hash}(_routing) + \text{hash}(_id) \% \text{routing_partition_size}) \% \text{num_primary_shards}$$

也就是说，_routing 字段用于计算索引中的一组分片，然后使用 _id 来选择该组内的分片。

index.routing_partition_size 取值应具有大于 1 且小于 index.number_of_shards 的值。

计算过程的实现如下：

```
private static int calculateScaledShardId(IndexMetaData indexMetaData,
String effectiveRouting, int partitionOffset) {
    final int hash = Murmur3HashFunction.hash(effectiveRouting) + partitionOffset;
    return Math.floorMod(hash, indexMetaData.getRoutingNumShards()) /
indexMetaData.getRoutingFactor();
}
```

effectiveRouting 是 id 或设置的 routing 值。partitionOffset 一般是 0。在设置了 index.routing_partition_size 的情况下其取值为：

```
partitionOffset = Math.floorMod(Murmur3HashFunction.hash(id),
indexMetaData.getRoutingPartitionSize());
```

indexMetaData.getRoutingNumShards() 的值为 routingNumShards，其取决于配置：index.number_of_routing_shards。如果没有配置，则 routingNumShards 的值等于 numberOfShards。

indexMetaData.getRoutingFactor()的值为：

```
routingNumShards / numberOfShards
```

numberOfShards 的值取决于配置：index.number_of_shards。

实现过程与公式稍有区别，最后多了一个值，这个值和索引拆分（split）过程有关，此处不详细讨论。

8. 转发请求并等待响应

主要是根据集群状态中的内容路由表确定主分片所在节点，转发请求并等待响应。

遍历所有需要写的 shard，将位于某个 shard 的请求封装为 BulkShardRequest 类，调用 TransportShardBulkAction#execute 执行发送，在 listener 中等待响应，每个响应也是以 shard 为单位的。如果某个 shard 的响应中部分 doc 写失败了，则将异常信息填充到 Response 中，整体请求做成功处理。

待收到所有响应后（无论成功还是失败的），回复给客户端。

转发请求的具体实现位于 TransportReplicationAction.ReroutePhase#doRun。

转发前先获取最新集群状态，根据集群状态中的内容路由表找到目的 shard 所在的主分片，如果主分片不在本机，则转发到相应的节点，否则在本地执行。

```
//获取主分片所在节点
final ShardRouting primary = primary(state);
final DiscoveryNode node = state.nodes().get(primary.currentNodeId());
//如果主分片在本节点，则在本地执行，否则转发出去
if (primary.currentNodeId().equals(state.nodes().getLocalNodeId())) {
    performLocalAction(state, primary, node, indexMetaData);
} else {
    performRemoteAction(state, primary, node);
}
```

7.4.2 主分片节点流程

执行本流程的线程池：bulk。

主分片所在节点负责在本地写主分片，写成功后，转发写副本片请求，等待响应，回复协调节点。

1. 检查请求

主分片所在节点收到协调节点发来的请求后也是先做了校验工作，主要检测要写的是不是



主分片, AllocationId (后续章节会介绍) 是否符合预期, 索引是否处于关闭状态等。

2. 是否延迟执行

判断请求是否需要延迟执行, 如果需要延迟则放入队列, 否则继续下面的流程。

3. 判断主分片是否已经发生迁移

如果已经发生迁移, 则转发请求到迁移的节点。

4. 检测写一致性

在开始写之前, 检测本次写操作涉及的 shard, 活跃 shard 数量是否足够, 不足则不执行写入。默认为 1, 只要主分片可用就执行写入。

```
public boolean enoughShardsActive(final IndexShardRoutingTable
shardRoutingTable) {
    final int activeShardCount = shardRoutingTable.activeShards().size();
    if (this == ActiveShardCount.ALL) {
        return activeShardCount == shardRoutingTable.replicaShards().size() + 1;
    } else if (this == ActiveShardCount.DEFAULT) {
        return activeShardCount >= 1;
    } else {
        return activeShardCount >= value;
    }
}
```

5. 写 Lucene 和事务日志

遍历请求, 处理动态更新字段映射, 然后调用 `InternalEngine#index` 逐条对 doc 进行索引。

Engine 封装了 Lucene 和 translog 的调用, 对外提供读写接口。

在写入 Lucene 之前, 先生成 Sequence Number 和 Version。这些都是在 InternalEngine 类中实现的。Sequence Number 每次递增 1, Version 根据当前 doc 的最大版本加 1。

索引过程为先写 Lucene, 后写 translog。

因为 Lucene 写入时对数据有检查, 写操作可能会失败。如果先写 translog, 写入 Lucene 时失败, 则还需要对 translog 进行回滚处理。

6. flush translog

根据配置的 translog flush 策略进行刷盘控制, 定时或立即刷盘:



```

private void maybeSyncTranslog(final IndexShard indexShard) throws
IOException {
    final Translog translog = indexShard.getTranslog();
    if (indexShard.getTranslogDurability() == Translog.Durability.REQUEST &&
        translog.getLastSyncedGlobalCheckpoint() < indexShard.
getGlobalCheckpoint()) {
        indexShard.getTranslog().sync();
    }
}

```

7. 写副分片

现在已经为要写的副本 shard 准备了一个列表，循环处理每个 shard，跳过 unassigned 状态的 shard，向目标节点发送请求，等待响应。这个过程是异步并行的。

转发请求时会将 SequenceID、PrimaryTerm、GlobalCheckPoint、version 等传递给副分片。

```

replicasProxy.performOn(shard, replicaRequest, globalCheckpoint,...);

```

在等待 Response 的过程中，本节点发出了多少个 Request，就要等待多少个 Response。无论这些 Response 是成功的还是失败的，直到超时。

收集到全部的 Response 后，执行 finish()。给协调节点返回消息，告知其哪些成功、哪些失败了。

```

private void decPendingAndFinishIfNeeded() {
    if (pendingActions.decrementAndGet() == 0) {
        finish();
    }
}

```

8. 处理副分片写失败情况

主分片所在节点将发送一个 shardFailed 请求给 Master:

```

replicasProxy.failShardIfNeeded(shard, message,
    replicaException, ReplicationOperation.this::decPendingAndFinishIfNeeded,
    ReplicationOperation.this::onPrimaryDemoted, throwable ->
decPendingAndFinishIfNeeded());

```

向 Master 发送 shardFailed 请求:




```
sendShardAction(SHARD_FAILED_ACTION_NAME, currentState, shardEntry, listener);
```

然后 Master 会更新集群状态，在新的集群状态中，这个 shard 将：

- 从 `in_sync_allocations` 列表中删除；
- 在 `routing_table` 的 shard 列表中将 state 由 `STARTED` 更改为 `UNASSIGNED`；
- 添加到 `routingNodes` 的 `unassignedShards` 列表。

7.4.3 副分片节点流程

执行本流程的线程池：`bulk`。

执行与主分片基本相同的写 doc 过程，写完后回复主分片节点。

```
protected void doRun() throws Exception {
    setPhase(task, "replica");
    final String actualAllocationId = this.replica.routingEntry().
allocationId().getId();
    //检查 AllocationId
    if (actualAllocationId.equals(targetAllocationID) == false) {
        throw new ShardNotFoundException();
    }
    replica.acquireReplicaOperationPermit(primaryTerm, globalCheckpoint,
this, executor);
}
```

在副分片的写入过程中，参数检查的实现与主分片略有不同，最终都调用 `IndexShard-OperationPermits#acquire` 判断是否需要 `delay`，继续后面的写流程。

7.5 I/O 异常处理

在一个 shard 上执行的一些操作可能会产生 I/O 异常之类的情况。一个 shard 上的 CRUD 等操作在 ES 里由一个 Engine 对象封装，在 Engine 处理过程中，部分操作产生的部分异常 ES 会认为有必要关闭此 Engine，上报 Master。例如，系统 I/O 层面的写入失败，这可能意味着磁盘损坏。

对 Engine 异常的捕获目前主要通过 `IOException` 实现。例如，索引文档过程中的异常处理：



```

try {
    //索引文档到 Lucene
    indexResult = indexIntoLucene(index, plan);
}
catch (RuntimeException | IOException e) {
    try {
        maybeFailEngine("index", e);
    } catch (Exception inner) {
        e.addSuppressed(inner);
    }
    throw e;
}

```

Engine 类中的 maybeFailEngine()负责检查是否应当关闭引擎 failEngine()。

可能会触发 maybeFailEngine()的操作如下表所示。

操 作	简 介	可能的异常
createSearcherManager	创建搜索管理器	IOException
index	索引文档	RuntimeException、IOException
delete	删除文档	RuntimeException、IOException
sync flush	同步刷新	IOException
sync commit	同步提交	IOException
flush	刷入磁盘	FlushFailedEngineException
force merge	手工合并 Lucene 分段	Exception

注意：其中不包含 get 操作，也就是说，读取 doc 失败不会触发 shard 迁移。

7.5.1 Engine 关闭过程

将 Lucene 标记为异常，简化的实现如下：

```

public void failEngine(String reason, @Nullable Exception failure) {
    failedEngine.set((failure != null) ? failure : new
IllegalStateException(reason));
    store.markStoreCorrupted(new IOException("failed engine (reason: [" +
reason + "]", failure));
}

```



关闭 shard，然后汇报给 Master：

```
private void failAndRemoveShard(...) {  
    //关闭 shard  
    indexService.removeShard(shardRouting.shardId().id(), message);  
    //向 Master 节点发送 SHARD_FAILED_ACTION_NAME 请求  
    sendFailShard(shardRouting, message, failure, state);  
}
```

7.5.2 Master 的对应处理

收到节点的 SHARD_FAILED_ACTION_NAME 消息后，Master 通过 reroute 将失败的 shard 通过 reroute 迁移到新的节点，并更新集群状态。

7.5.3 异常流程总结

(1) 如果请求在协调节点的路由阶段失败，则会等待集群状态更新，拿到更新后，进行重试，如果再次失败，则仍旧等集群状态更新，直到超时 1 分钟为止。超时后仍失败则进行整体请求失败处理。

(2) 在主分片写入过程中，写入是阻塞的。只有写入成功，才会发起写副本请求。如果主 shard 写失败，则整个请求被认为处理失败。如果有部分副本写失败，则整个请求被认为处理成功。

(3) 无论主分片还是副分片，当写一个 doc 失败时，集群不会重试，而是关闭本地 shard，然后向 Master 汇报，删除是以 shard 为单位的。

7.6 系统特性

ES 本身也是一个分布式存储系统，如同其他分布式系统一样，我们经常关注的一些特性如下。

- 数据可靠性：通过分片副本和事务日志机制保障数据安全。
- 服务可用性：在可用性和一致性的取舍方面，默认情况下 ES 更倾向于可用性，只要主分片可用即可执行写入操作。
- 一致性：笔者认为是弱一致性。只要主分片写成功，数据就可能被读取。因此读取操作在主分片和副分片上可能会得到不同结果。



- 原子性：索引的读写、别名更新是原子操作，不会出现中间状态。但 bulk 不是原子操作，不能用来实现事务。
- 扩展性：主副分片都可以承担读请求，分担系统负载。

7.7 思考

分析完写入流程后，也许读者已经意识到了这个过程的一些缺点：

- 副分片写入过程需要重新生成索引，不能单纯复制数据，浪费计算能力，影响入库速度。
- 磁盘管理能力较差，对坏盘检查和容忍性比 HDFS 差不少。例如，在配置多磁盘路径的情况下，有一块坏盘就无法启动节点。



8 chapter

第 8 章 GET 流程

ES 的读取分为 GET 和 Search 两种操作，这两种读取操作有较大的差异，GET/MGET 必须指定三元组：_index、_type、_id。也就是说，根据文档 id 从正排索引中获取内容。而 Search 不指定_id，根据关键词从倒排索引中获取内容。本章分析 GET/MGET 过程，下一章分析 Search 过程。

一个 GET 请求的简单例子（来自官网）如下：

```
curl -XGET http://127.0.0.1:9200/website/blog/1?pretty
{
  "_index" : "website",
  "_type" : "blog",
  "_id" : "1",
  "_version" : 21,
  "found" : true,
  "_source" : {
    "first_name" : "John",
    "last_name" : "Smith",
    "age" : 25,
    "about" : "I love to go rock climbing",
    "interests" : [
      "sports",
      "music"
    ]
  }
}
```



8.1 可选参数

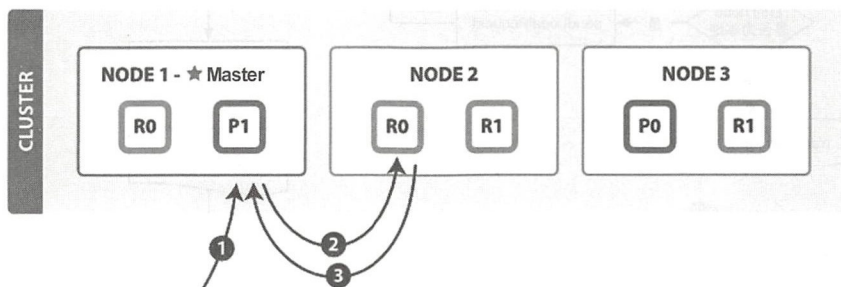
与写请求相同，GET 请求时可以在 URI 中设置一些可选参数，如下表所示。

参 数	简 介
realtime	默认为 true。GET API 默认是实时的，不受索引刷新（refresh）频率设置的影响。如果文档已经更新，但还没有刷新，则 GET API 将会发出一个刷新调用，使文档可见
source filtering	默认情况下，GET API 返回文档的全部内容（在 <code>_source</code> 字段中）。可以设置为 false，不返回文档内容。同时可以使用 <code>_source_include</code> 和 <code>_source_exclude</code> 过滤返回原始文档的部分字段
stored Fields	对于索引映射中 store 设置为 true 的字段，本选项用来指定返回哪些字段
_source	通过 <code>{index}/{type}/{id}/_source</code> 的形式可以只返回原始文档内容，其他的 <code>_id</code> 等元信息不返回
routing	自定义 routing
preference	默认情况下，GET API 从分片的多个副本中随机选择一个，通过指定优先级（preference）可以选择从主分片读取，或者尝试从本地读取
refresh	默认为 false，若设置 refresh 为 true，则可以在读取之前先执行刷新操作，这对写入速度有负面影响
version	如果 GET API 指定了版本号，那么当文档实际版本号与请求的不符时，ES 将返回 409 错误

8.2 GET 基本流程

搜索和读取文档都属于读操作，可以从主分片或副分片中读取数据。

读取单个文档的流程（图片来自官网）如下图所示。



这个例子中的索引有一个主分片和两个副分片。以下是从主分片或副分片中读取时的步骤：



(1) 客户端向 NODE1 发送读请求。

(2) NODE1 使用文档 ID 来确定文档属于分片 0，通过集群状态中的内容路由表信息获知分片 0 有三个副本数据，位于所有的三个节点中，此时它可以将请求发送到任意节点，这里它将请求转发到 NODE2。

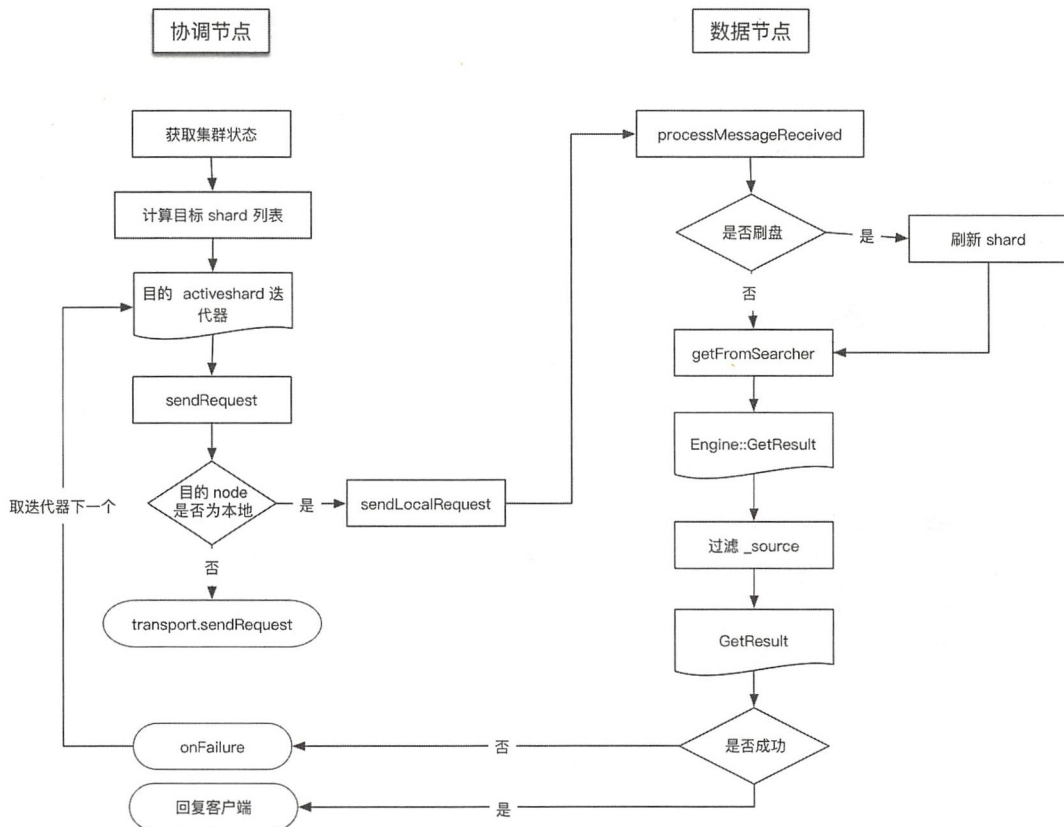
(3) NODE2 将文档返回给 NODE1，NODE1 将文档返回给客户端。

NODE1 作为协调节点，会将客户端请求轮询发送到集群的所有副本来实现负载均衡。

在读取时，文档可能已经存在于主分片上，但还没有复制到副分片。在这种情况下，读请求命中副分片时可能会报告文档不存在，但是命中主分片可能成功返回文档。一旦写请求成功返回给客户端，则意味着文档在主分片和副分片都是可用的。

8.3 GET 详细分析

GET/MGET 流程涉及两个节点：协调节点和数据节点，流程如下图所示。



8.3.1 协调节点

执行本流程的线程池：`http_server_worker`。

`TransportSingleShardAction` 类用来处理存在于一个单个（主或副）分片上的读请求。将请求转发到目标节点，如果请求执行失败，则尝试转发到其他节点读取。在收到读请求后，处理过程如下。

内容路由

（1）在 `TransportSingleShardAction.AsyncSingleAction` 构造函数中，准备集群状态、节点列表等信息。

（2）根据内容路由算法计算目标 `shardid`，也就是文档应该落在哪个分片上。

（3）计算出目标 `shardid` 后，结合请求参数中指定的优先级和集群状态确定目标节点，由于分片可能存在多个副本，因此计算出的是一个列表。

```
private AsyncSingleAction(Request request, ActionListener<Response> listener) {
    ClusterState clusterState = clusterService.state();

    //集群 nodes 列表
    nodes = clusterState.nodes();

    //解析请求，更新自定义 routing
    resolveRequest(clusterState, internalRequest);
    //根据路由算法计算得到目的 shard 迭代器，或者根据优先级选择目标节点
    this.shardIt = shards(clusterState, internalRequest);
}
```

具体的路由算法参考写流程分析。

转发请求

作为协调节点，向目标节点转发请求，或者目标是本地节点，直接读取数据。发送函数声明了如何对 `Response` 进行处理：`AsyncSingleAction` 类中声明对 `Response` 进行处理的函数。无论请求在本节点处理还是发送到其他节点，均对 `Response` 执行相同的处理逻辑：

```
private void perform(@Nullable final Exception currentFailure) {
    DiscoveryNode node = nodes.get(shardRouting.currentNodeId());
    if (node == null) {
```

```

        onFailure(shardRouting, new NoShardAvailableActionException
(shardRouting.shardId()));
    } else {
        internalRequest.request().internalShardId = shardRouting.shardId();
        transportService.sendRequest(node, ...
            public void handleResponse(final Response response) {
                listener.onResponse(response);
            }
            public void handleException(TransportException exp) {
                onFailure(shardRouting, exp);
            }
        });
    }
}

```

发送的具体过程:

(1) 在 `TransportService::sendRequest` 中检查目标是否是本地 `node`。

(2) 如果是本地 `node`, 则进入 `TransportService#sendLocalRequest` 流程, `sendLocalRequest` 不发送到网络, 直接根据 `action` 获取注册的 `reg`, 执行 `processMessageReceived`:

```

private void sendLocalRequest(long requestId, final String action, final
TransportRequest request, TransportRequestOptions options) {
    final DirectResponseChannel channel = new DirectResponseChannel(logger,
localNode, action, requestId, this, threadPool);
    try {
        //根据 action 获取注册的 reg
        final RequestHandlerRegistry reg = getRequestHandler(action);
        reg.processMessageReceived(request, channel);
    }
}

```

(3) 如果发送到网络, 则请求被异步发送, “`sendRequest`” 的时候注册 `handle`, 等待处理 `Response`, 直到超时。

(4) 等待数据节点的回复, 如果数据节点处理成功, 则返回给客户端; 如果数据节点处理失败, 则进行重试:

```

private void onFailure(ShardRouting shardRouting, Exception e) {

```

```
perform(e);
}
```

内容路由结束时构造了目标节点列表的迭代器，重试发送时，目标节点选择迭代器的下一个。

8.3.2 数据节点

执行本流程的线程池：`get`。

数据节点接收协调节点请求的入口为：`TransportSingleShardAction.ShardTransportHandler#messageReceived`。

读取数据并组织成 `Response`，给客户端 `channel` 返回：

```
public void messageReceived(final Request request, final TransportChannel
channel) throws Exception {
    Response response = shardOperation(request, request.internalShardId);
    channel.sendResponse(response);
}
```

`shardOperation` 先检查是否需要 `refresh`，然后调用 `indexShard.getService().get()` 读取数据并存储到 `GetResult` 中。

读取及过滤

在 `ShardGetService#get()` 函数中，调用：

```
GetResult getResult = innerGet();
```

获取结果。`GetResult` 类用于存储读取的真实数据内容。核心的数据读取实现在 `ShardGetService#innerGet()` 函数中：

```
private GetResult innerGet(...) {
    final Collection<String> types;
    //处理_all选项
    if (type == null || type.equals("_all")) {
        ...
    }
    Engine.GetResult get = null;
    for (String typeX : types) {
        //调用 Engine 读取数据
```



```

        get = indexShard.get(new Engine.Get(realtime, typeX, id, uidTerm)
            .version(version).versionType(versionType));
    }
    try {
        //过滤返回结果
        return innerGetLoadFromStoredFields(type, id, gFields,
            fetchSourceContext, get, mapperService);
    } finally {
        get.release();
    }
}

```

(1) 通过 `indexShard.get()` 获取 `Engine.GetResult`。`Engine.GetResult` 类与 `innerGet` 返回的 `GetResult` 是同名的类，但实现不同。`indexShard.get()` 最终调用 `InternalEngine#get` 读取数据。

(2) 调用 `ShardGetService#innerGetLoadFromStoredFields()`，根据 `type`、`id`、`DocumentMapper` 等信息从刚刚获取的信息中获取数据，对指定的 `field`、`source` 进行过滤（`source` 过滤只支持对字段），把结果存于 `GetResult` 对象中。

InternalEngine 的读取过程

`InternalEngine#get` 过程会加读锁。处理 `realtime` 选项，如果为 `true`，则先判断是否有数据可以刷盘，然后调用 `Searcher` 进行读取。`Searcher` 是对 `IndexSearcher` 的封装。

在早期的 ES 版本中，如果开启（默认）`realtime`，则会尝试从 `translog` 中读取，刚写入不久的数据可以从 `translog` 中读取；从 ES 5.x 开始不会从 `translog` 中读取，只从 `Lucene` 中读。`realtime` 的实现机制变成依靠 `refresh` 实现。参考官方链接：<https://github.com/elastic/elasticsearch/pull/20102>。

```

public GetResult get(Get get, BiFunction<String, SearcherScope, Searcher>
    searcherFactory) throws EngineException {
    try (ReleasableLock ignored = readLock.acquire()) {
        ensureOpen();
        SearcherScope scope;
        //处理 realtime 选项，判断是否需要刷盘
        if (get.realtime()) {
            //versionMap 中的值是写入索引的时候添加的，不会写磁盘
            VersionValue versionValue = versionMap.getUnderLock(
                get.uid().bytes());
            if (versionValue != null) {
                if (versionValue.isDelete()) {

```

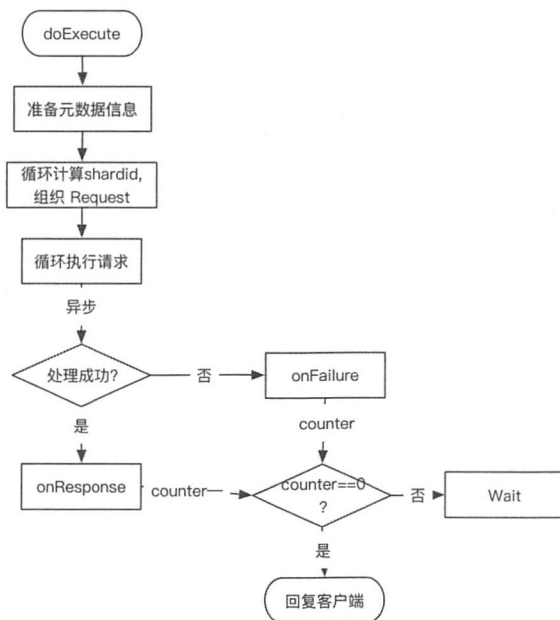
```

        return GetResult.NOT_EXISTS;
    }
    if (get.versionType().isVersionConflictForReads
(versionValue.version, get.version())) {
        throw new VersionConflictEngineException(...);
    }
    //执行刷盘操作
    refresh("realtime_get", SearcherScope.INTERNAL);
}
scope = SearcherScope.INTERNAL;
} else {
    scope = SearcherScope.EXTERNAL;
}
//调用 Searcher 读取数据
return getFromSearcher(get, searcherFactory, scope);
}
}

```

8.4 MGET 流程分析

MGET 的主要处理类：TransportMultiGetAction，通过封装单个 GET 请求实现，处理流程如下图所示。



主要流程如下：

(1) 遍历请求，计算出每个 doc 的路由信息，得到由 shardid 为 key 组成的 request map。这个过程没有在 TransportSingleShardAction 中实现，是因为如果在那里实现，shardid 就会重复，这也是合并为基于分片的请求的过程。

(2) 循环处理组织好的每个 shard 级请求，调用处理 GET 请求时使用 TransportSingleShardAction#AsyncSingleAction 处理单个 doc 的流程。

(3) 收集 Response，全部 Response 返回后执行 finishHim()，给客户端返回结果。

回复的消息中文档顺序与请求的顺序一致。如果部分文档读取失败，则不影响其他结果，检索失败的 doc 会在回复信息中标出。

8.5 思考

我们需要警惕实时读取特性，GET API 默认是实时的，实时的意思是写完了可以立刻读取，但仅限于 GET、MGET 操作，不包括搜索。在 5.x 版本之前，GET/MGET 的实时读取依赖于从 translog 中读取实现，5.x 版本之后的版本改为 refresh，因此系统对实时读取的支持会对写入速度有负面影响。

由此引出另一个较深层次的问题是，update 操作需要先 GET 再写，为了保证一致性，update 调用 GET 时将 realtime 选项设置为 true，并且不可配置。因此 update 操作可能会导致 refresh 生成新的 Lucene 分段。

读失败是怎么处理的？ 尝试从别的分片副本读取。

优先级 优先级策略只是将匹配到优先级的节点放到了目标节点列表的前面。

9 chapter

第 9 章

Search 流程

GET 操作只能对单个文档进行处理，由 `_index`、`_type` 和 `_id` 三元组来确定唯一文档。但搜索需要一种更复杂的模型，因为不知道查询会命中哪些文档。

找到匹配文档仅仅完成了搜索流程的一半，因为多分片中的结果必须组合成单个排序列表。集群的任意节点都可以接收搜索请求，接收客户端请求的节点称为协调节点。在协调节点，搜索任务被执行成一个两阶段过程，即 `query then fetch`。真正执行搜索任务的节点称为数据节点。

需要两个阶段才能完成搜索的原因是，在查询的时候不知道文档位于哪个分片，因此索引的所有分片（某个副本）都要参与搜索，然后协调节点将结果合并，再根据文档 ID 获取文档内容。例如，有 5 个分片，查询返回前 10 个匹配度最高的文档，那么每个分片都查询出当前分片的 TOP 10，协调节点将 $5 \times 10 = 50$ 的结果再次排序，返回最终 TOP 10 的结果给客户端。

一个简单的搜索请求示例如下：

```
curl -XGET "localhost:9200/_search?q=first&pretty"
{
  "took" : 3,
  "timed_out" : false,
  "_shards" : {
    "total" : 6,
    "successful" : 6,
    "skipped" : 0,
    "failed" : 0
  },
}
```

```
"hits" : {
  "total" : 1,
  "max_score" : 0.2876821,
  "hits" : [
    {
      "_index" : "website",
      "_type" : "blog",
      "_id" : "2",
      "_score" : 0.2876821,
      "_source" : {
        "title" : "My first blog entry",
        "text" : "Just trying this out..."
      }
    }
  ]
}
```

在上面的例子中，我们从所有字段搜索“first”关键词，返回信息中几个基本字段的含义如下：

- took 代表搜索执行时间（单位：毫秒）；
- total 代表本次搜索命中的文档数量；
- max_score 为最大得分，代表文档匹配度；
- hits 为搜索命中的结果列表，默认为 10 条。

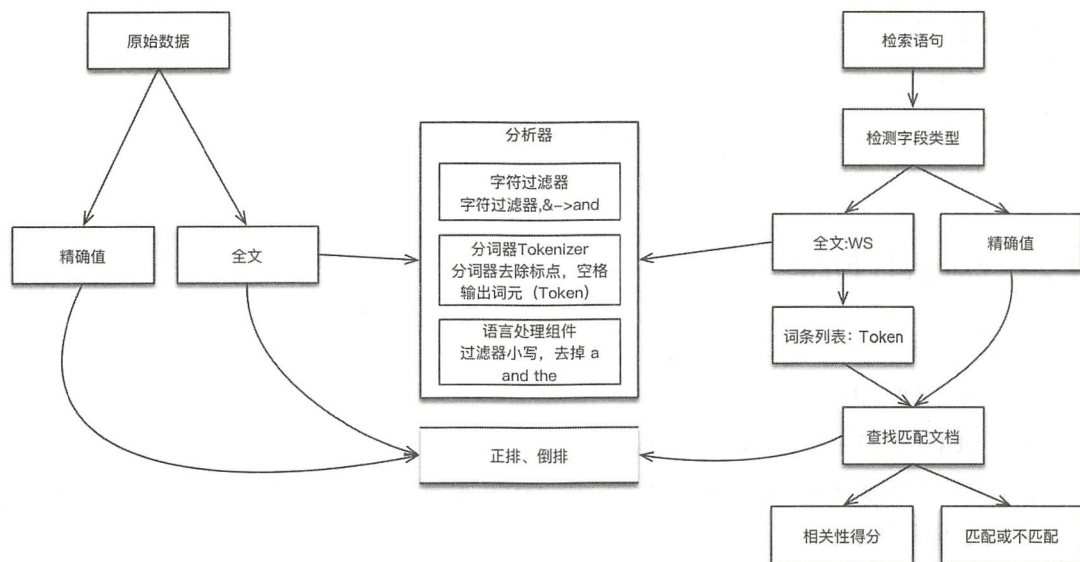
9.1 索引和搜索

ES 中的数据可以分为两类：精确值和全文。

- 精确值，比如日期和用户 id、IP 地址等。
- 全文，指文本内容，比如一条日志，或者邮件的内容。

这两种类型的数据在查询时是不同的：对精确值的比较是二进制的，查询要么匹配，要么不匹配；全文内容的查询无法给出“有”还是“没有”的结果，它只能找到结果是“看起来像”你要查询的东西，因此把查询结果按相似度排序，评分越高，相似度越大。

对数据建立索引和执行搜索的原理如下图所示。



9.1.1 建立索引

如果是全文数据，则对文本内容进行分析，这项工作在 ES 中由分析器实现。分析器实现如下功能：

- 字符过滤器。主要是对字符串进行预处理，例如，去掉 HTML，将&转换成 and 等。
- 分词器 (Tokenizer)。将字符串分割为单个词条，例如，根据空格和标点符号分割，输出的词条称为词元 (Token)。
- Token 过滤器。根据停止词 (Stop word) 删除词元，例如，and、the 等无用词，或者根据同义词表增加词条，例如，jump 和 leap。
- 语言处理。对上一步得到的 Token 做一些和语言相关的处理，例如，转为小写，以及将单词转换为词根的形式。语言处理组件输出的结果称为词 (Term)。

分析完毕后，将分析器输出的词 (Term) 传递给索引组件，生成倒排和正排索引，再存储到文件系统中。

9.1.2 执行搜索

搜索调用 Lucene 完成，如果是全文检索，则：

- 对检索字段使用建立索引时相同的分析器进行分析，产生 Token 列表；

- 根据查询语句的语法规则转换成一棵语法树；
- 查找符合语法树的文档；
- 对匹配到的文档列表进行相关性评分，评分策略一般使用 TF/IDF；
- 根据评分结果进行排序。

9.2 search type

ES 目前有两种搜索类型：

- DFS_QUERY_THEN_FETCH；
- QUERY_THEN_FETCH（默认）。

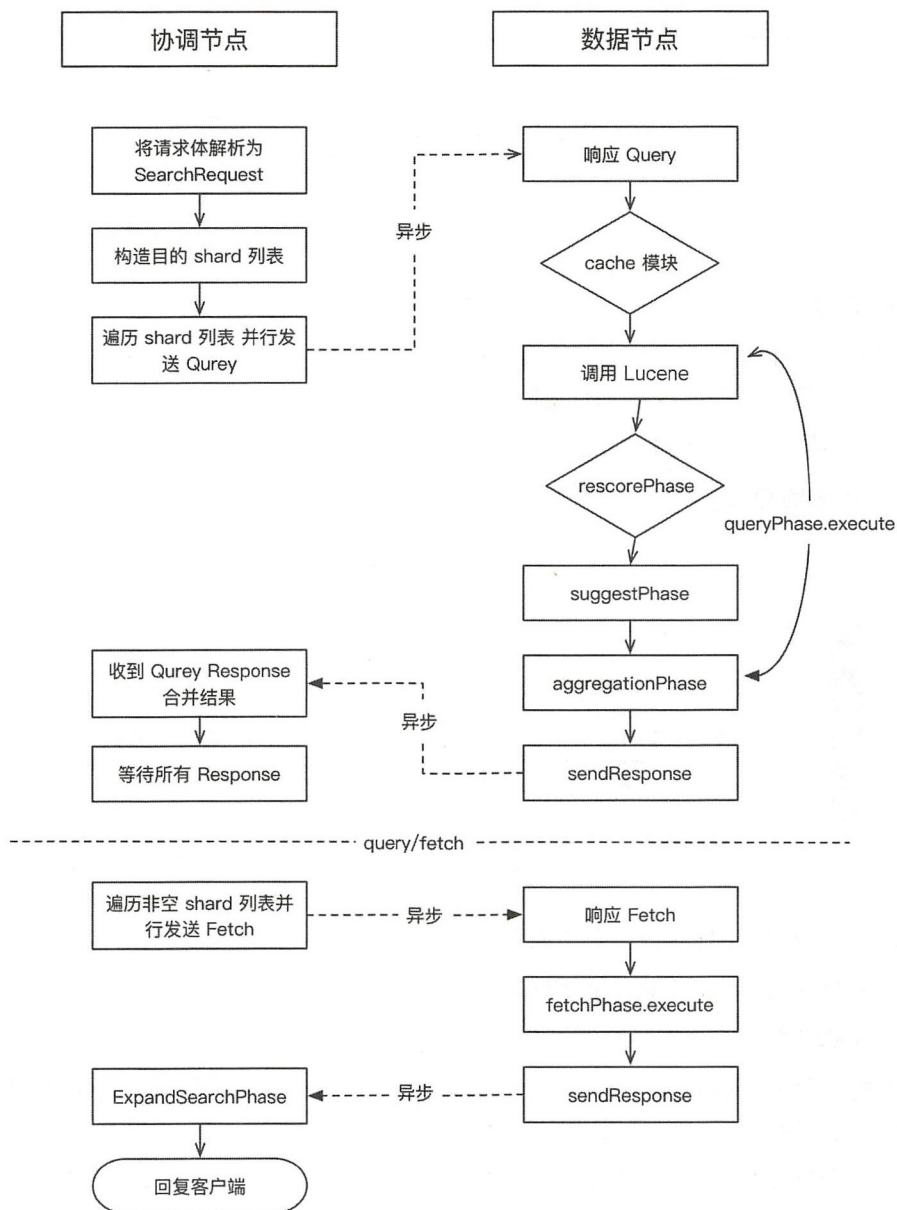
两种不同的搜索类型的区别在于查询阶段，DFS 查询阶段的流程要多一些，它使用全局信息来获取更准确的评分。

本章的流程分析默认搜索类型。下面我们仍旧按照请求涉及的节点来分析流程，搜索流程涉及两个节点：协调节点和数据节点。

9.3 分布式搜索过程

一个搜索请求必须询问请求的索引中所有分片的某个副本来进行匹配。假设一个索引有 5 个主分片，每个主分片有 1 个副分片，共 10 个分片，一次搜索请求会由 5 个分片来共同完成，它们可能是主分片，也可能是副分片。也就是说，一次搜索请求只会命中所有分片副本中的一个。

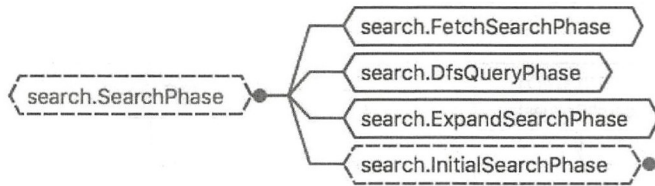
当搜索任务执行在分布式系统上时，整体流程如下图所示。



9.3.1 协调节点流程

两阶段相应的实现位置：查询（Query）阶段——`search.InitialSearchPhase`；取回（Fetch）阶段——`search.FetchSearchPhase`。

它们都继承自 SearchPhase，如下图所示。

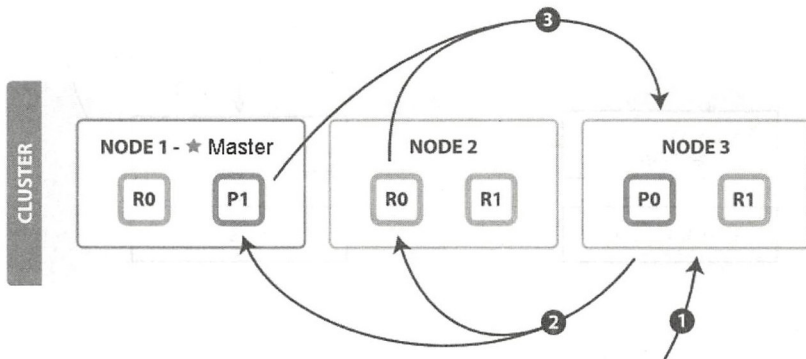


Query 阶段

在初始查询阶段，查询会广播到索引中每一个分片副本（主分片或副分片）。每个分片在本地执行搜索并构建一个匹配文档的优先队列。

优先队列是一个存有 topN 匹配文档的有序列表。优先队列大小为分页参数 from + size。

分布式搜索的 Query 阶段（图片来自官网）如下图所示。



QUERY_THEN_FETCH 搜索类型的查询阶段步骤如下：

- (1) 客户端发送 search 请求到 NODE 3。
- (2) Node 3 将查询请求转发到索引的每个主分片或副分片中。
- (3) 每个分片在本地执行查询，并使用本地的 Term/Document Frequency 信息进行打分，添加结果到大小为 from + size 的本地有序优先队列中。

(4) 每个分片返回各自优先队列中所有文档的 ID 和排序值给协调节点，协调节点合并这些值到自己的优先队列中，产生一个全局排序后的列表。

协调节点广播查询请求到所有相关分片时，可以是主分片或副分片，协调节点将在之后的请求中轮询所有的分片副本来分摊负载。

查询阶段并不会对搜索请求的内容进行解析，无论搜索什么内容，只看本次搜索需要命中哪些 shard，然后针对每个特定 shard 选择一个副本，转发搜索请求。

Query 阶段源码分析

执行本流程的线程池：http_server_work。

1. 解析请求

在 RestSearchAction#prepareRequest 方法中将请求体解析为 SearchRequest 数据结构：

```
public RestChannelConsumer prepareRequest(...)
{
    SearchRequest searchRequest = new SearchRequest();
    request.withContentOrSourceParamParserOrNull(parser ->
        parseSearchRequest(searchRequest, request, parser, setSize));
}
```

2. 构造目的 shard 列表

将请求涉及的本集群 shard 列表和远程集群的 shard 列表(远程集群用于跨集群访问)合并：

```
private void executeSearch(...)
{
    GroupShardsIterator<ShardIterator> localShardsIterator =
clusterService.operationRouting().searchShards(clusterState,
        concreteIndices, routingMap, searchRequest.preference(),
searchService.getResponseCollectorService(), nodeSearchCounts);
    GroupShardsIterator<SearchShardIterator> shardIterators =
mergeShardsIterators(localShardsIterator, localIndices,
        remoteShardIterators);
}
```

3. 遍历所有 shard 发送请求

请求是基于 shard 遍历的，如果列表中有 N 个 shard 位于同一个节点，则向其发送 N 次请求，并不会把请求合并为一个。

```
public final void run() throws IOException {
    if (shardsIts.size() > 0) {
        //最大发分片请求数可以通过 max_concurrent_shard_requests 参数配置(v6.0 新增)
        int maxConcurrentShardRequests = Math.min(this.maxConcurrentShardRequests,
shardsIts.size());
```



```

        for (int index = 0; index < maxConcurrentShardRequests; index++) {
            final SearchShardIterator shardRoutings = shardsIts.get(index);
            //执行 shard 级请求
            performPhaseOnShard(index, shardRoutings, shardRoutings.
nextOrNull());
        }
    }
}

```

`shardsIts` 为本次搜索涉及的所有分片，`shardRoutings.nextOrNull()`从某个分片的所有副本中选择一个，例如，从 `website` 中选择主分片。

转发请求同时定义一个 `Listener`，用于处理 `Response`：

```

private void performPhaseOnShard(...) {
    executePhaseOnShard(...) {
        //收到执行成功的回复
        public void innerOnResponse(FirstResult result) {
            maybeFork(thread, () -> onShardResult(result, shardIt));
        }
        //收到执行失败的回复
        public void onFailure(Exception t) {
            maybeFork(thread, () -> onShardFailure(shardIndex, shard,
shard.currentNodeId(), shardIt, t));
        }
    });
}

```

发送过程依然调用 `transport` 模块实现。

4. 收集返回结果

本过程在 `search` 线程池中执行：

```

private void onShardResult(FirstResult result, SearchShardIterator shardIt) {
    onShardSuccess(result);
    successfulShardExecution(shardIt);
}

```

`onShardSuccess` 对收集到的结果进行合并。`successfulShardExecution` 方法检查是否所有请求

都已收到回复，是否进入下一阶段：

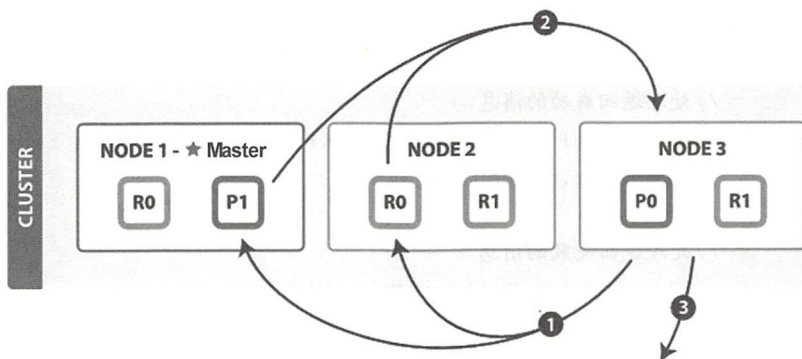
```
private void successfulShardExecution(SearchShardIterator shardsIt) {
    //计数器累加
    final int xTotalOps = totalOps.addAndGet(remainingOpsOnIterator);
    //检查是否收到全部回复
    if (xTotalOps == expectedTotalOps) {
        onPhaseDone();
    } else if (xTotalOps > expectedTotalOps) {
        throw new AssertionError(...);
    }
}
```

onPhaseDone 会调用 executeNextPhase，从而开始执行取回阶段。

Fetch 阶段

Query 阶段知道了要取哪些数据，但是并没有取具体的数据，这就是 Fetch 阶段要做的。

分布式搜索的 Fetch 阶段（图片来自官网）如下图所示。



Fetch 阶段由以下步骤构成：

- (1) 协调节点向相关 NODE 发送 GET 请求。
- (2) 分片所在节点向协调节点返回数据。
- (3) 协调节点等待所有文档被取得，然后返回给客户端。

分片所在节点在返回文档数据时，处理有可能出现的_source 字段和高亮参数。

协调节点首先决定哪些文档“确实”需要被取回，例如，如果查询指定了 { "from": 90, "size": 10 }，则只有从第 91 个开始的 10 个结果需要被取回。

为了避免在协调节点中创建的 `number_of_shards * (from + size)` 优先队列过大，应尽量控制分页深度。

Fetch 阶段源码解析

Fetch 阶段的目的是通过文档 ID 获取完整的文档内容。

执行本流程的线程池：`search`。

1. 发送 Fetch 请求

Query 阶段的 `executeNextPhase` 方法触发 Fetch 阶段，Fetch 阶段的起点为 `FetchSearchPhase#innerRun` 函数，从查询阶段的 `shard` 列表中遍历，跳过查询结果为空的 `shard`，对特定目标 `shard` 执行 `executeFetch` 来获取数据，其中包括分页信息。对 `scroll` 请求的处理也在 `FetchSearchPhase#innerRun` 函数中。

`executeFetch` 的主要实现：

```
private void executeFetch(...)
{
    //发送请求
    context.getSearchTransport().sendExecuteFetch(connection,
fetchSearchRequest, context.getTask(),
        new SearchActionListener<FetchSearchResult>(shardTarget, shardIndex) {
            //处理返回成功的消息
            public void innerOnResponse(FetchSearchResult result) {
                counter.onResult(result);
            }
            //处理返回失败的消息
            public void onFailure(Exception e) {
                try {
                    counter.onFailure(shardIndex, shardTarget, e);
                } finally {
                    releaseIrrelevantSearchContext(querySearchResult);
                }
            }
        }
    );
}
```

`executeFetch` 的参数 `querySearchResult` 中包含分页信息，最后定义一个 `Listener`，每成功获取一个 `shard` 数据后就执行 `counter.onResult`，其中调用对结果的处理回调，把 `result` 保存到数组



中，然后执行 `countDown`:

```
void onResult(R result) {
    try {
        resultConsumer.accept(result);
    } finally {
        countDown();
    }
}
```

2. 收集结果

收集器的定义在 `innerRun` 中，包括收到的 `shard` 数据存放在哪里，收集完成后谁来处理:

```
final CountedCollector<FetchSearchResult> counter = new CountedCollector<>
(r -> fetchResults.set(r.getShardIndex(), r),
    docIdsToLoad.length,
    finishPhase, context);
```

`fetchResults` 用于存储从某个 `shard` 收集到的结果，每收到一个 `shard` 的数据就执行一次 `counter.countDown()`。当所有 `shard` 数据收集完毕后，`countDown` 会触发执行 `finishPhase`:

```
final Runnable finishPhase = ()
    -> moveToNextPhase(searchPhaseController, scrollId,
reducedQueryPhase, queryAndFetchOptimization ?
    queryResults : fetchResults);
```

`moveToNextPhase` 方法执行下一阶段，下一阶段要执行的任务定义在 `FetchSearchPhase` 构造函数中，主要是触发 `ExpandSearchPhase`:

```
FetchSearchPhase(InitialSearchPhase.SearchPhaseResults<SearchPhaseResult>
    resultConsumer, SearchPhaseController searchPhaseController,
    SearchPhaseContext context) {
    this(resultConsumer, searchPhaseController, context,
        (response, scrollId) -> new ExpandSearchPhase(context, response,
// collapse only happens if the request has inner hits
        (finalResponse) -> sendResponsePhase(finalResponse, scrollId,
context)));
}
```



3. ExpandSearchPhase

取回阶段完成之后执行 `ExpandSearchPhase#run`，主要判断是否启用字段折叠，根据需要实现字段折叠功能，如果没有实现字段折叠，则直接返回给客户端。

4. 回复客户端

`ExpandSearchPhase` 执行完之后回复客户端，在 `sendResponsePhase` 方法中实现：

```
private static SearchPhase sendResponsePhase(...) {
    return new SearchPhase("response") {
        public void run() throws IOException {
            context.onResponse(context.buildSearchResponse(response, scrollId));
        }
    };
}
```

9.3.2 执行搜索的数据节点流程

执行本流程的线程池：`search`。

对各种 `Query`、`Fetch` 请求的处理入口注册于 `SearchTransportService#registerRequestHandler`。

响应 `Query` 请求

以常见的 `Query` 请求为例，其 `action` 为：

```
indices:data/read/search[phase/query]
```

主要过程就是执行查询，然后发送 `Response`：

```
transportService.registerRequestHandler(QUERY_ACTION_NAME,
ShardSearchTransportRequest::new, ThreadPool.Names.SAME,
    new TaskAwareTransportRequestHandler<ShardSearchTransportRequest>() {
        //收到 Query 请求
        public void messageReceived(...) throws Exception {
            //执行查询
            searchService.executeQueryPhase(request, (SearchTask) task,
new ActionListener<SearchPhaseResult>() {
                //处理查询成功的情况
                public void onResponse(SearchPhaseResult searchPhaseResult) {
```




```
channel.sendResponse(searchPhaseResult);

    }
    //处理查询失败的情况
    public void onFailure(Exception e) {
        channel.sendResponse(e);
    }
    });
}
```

查询实现入口在 `searchService.executeQueryPhase` 中（完全封装在这个方法中）。查询时，先看是否允许 `cache`，由以下配置决定：

```
index.requests.cache.enable
```

默认为 `true`，会把查询结果放到 `cache` 中，查询时优先从 `cache` 中取。这个 `cache` 由节点的所有分片共享，基于 LRU 算法实现：空间满的时候删除最近最少使用的数据。`cache` 并不缓存全部检索结果。

核心的查询封装在 `queryPhase.execute(context)` 中，其中调用 Lucene 实现检索，同时实现聚合：

```
public void execute(SearchContext searchContext){
    aggregationPhase.preProcess(searchContext);
    boolean rescore = execute(searchContext, searchContext.searcher(),
searcher::setCheckCancelled, indexSort);
    if (rescore) {
        rescorePhase.execute(searchContext);
    }
    suggestPhase.execute(searchContext);
    aggregationPhase.execute(searchContext);
}
```

其中包含几个核心功能：

- `execute()`，调用 Lucene、`searcher.search()` 实现搜索；
- `rescorePhase`，全文检索且需要打分；
- `suggestPhase`，自动补全及纠错；



- aggregationPhase, 实现聚合。

总结:

- 慢查询 Query 日志的统计时间在于本阶段的处理时间。
- 聚合操作在本阶段实现, 在 Lucene 检索后完成。

响应 Fetch 请求

以常见的基于 id 进行 Fetch 请求为例, 其 action 为:

```
indices:data/read/search[phase/fetch/id]
```

主要过程是执行 Fetch, 然后发送 Response:

```
transportService.registerRequestHandler(FETCH_ID_ACTION_NAME,
ShardFetchSearchRequest::new, ThreadPool.Names.SEARCH,
    new TaskAwareTransportRequestHandler<ShardFetchSearchRequest>() {
        //收到 Fetch 请求
        public void messageReceived(...) throws Exception {
            //执行 Fetch
            FetchSearchResult result = searchService.executeFetchPhase
(request, (SearchTask)task);
            channel.sendResponse(result);
        }
    });
```

对 Fetch 响应的实现封装在 searchService.executeFetchPhase 中, 核心是调用 fetchPhase.execute(context)。按照命中的 doc 取得相关数据, 填充到 SearchHits 中, 最终封装到 FetchSearchResult 中。

总结:

慢查询 Fetch 日志的统计时间在于本阶段的处理时间。

9.4 小结

- 聚合是在 ES 中实现的, 而非 Lucene。
- Query 和 Fetch 请求之间是无状态的, 除非是 scroll 方式。
- 分页搜索不会单独 “cache”, cache 和分页没有关系。



- 每次分页的请求都是一次重新搜索的过程，而不是从第一次搜索的结果中获取。看上去不太符合常规的做法，事实上互联网的搜索引擎都是重新执行了搜索过程：人们基本只看前几页，很少深度分页；重新执行一次搜索很快；如果缓存第一次搜索结果等待翻页命中，则这种缓存的代价较大，意义却不大，因此不如重新执行一次搜索。
- 搜索需要遍历分片所有的 Lucene 分段，因此合并 Lucene 分段对搜索性能有好处。



10 chapter

第 10 章 索引恢复流程分析

索引恢复（indices.recovery）是 ES 数据恢复过程。待恢复的数据是客户端写入成功，但未执行刷盘（flush）的 Lucene 分段。例如，当节点异常重启时，写入磁盘的数据先到文件系统的缓冲，未必来得及刷盘，如果不通过某种方式将未刷盘的数据找回来，则会丢失一些数据，这是保持数据完整性的体现；另一方面，由于写入操作在多个分片副本上没有来得及全部执行，副分片需要同步成和主分片完全一致，这是数据副本一致性的体现。

根据数据分片性质，索引恢复过程可分为主分片恢复流程和副分片恢复流程。

- 主分片从 translog 中自我恢复，尚未执行 flush 到磁盘的 Lucene 分段可以从 translog 中重建；
- 副分片需要从主分片中拉取 Lucene 分段和 translog 进行恢复。但是有机会跳过拉取 Lucene 分段的过程。

索引恢复的触发条件包括从快照备份恢复、节点加入和离开、索引的 _open 操作等。

恢复工作一般经历以下几个阶段（stage），如下表所示。

阶 段	简 介
INIT	恢复尚未启动
INDEX	恢复 Lucene 文件，以及在节点间复制索引数据
VERIFY_INDEX	验证索引
TRANSLOG	启动 engine，重放 translog，建立 Lucene 索引
FINALIZE	清理工作
DONE	完毕



主分片和副分片恢复都会经历这些阶段，但有时候会跳过具体执行过程，只是在流程上体现出经历了这个短暂阶段。例如，副分片恢复时会跳过 TRANSLOG 重放过程；主分片恢复过程中的 INDEX 阶段不会在节点之间复制数据。

10.1 相关配置

索引恢复流程有以下配置项，都支持动态调整，如下表所示。

配 置	简 介
indices.recovery.max_bytes_per_sec	副分片恢复的 phase1 过程中，主副分片节点之间传输数据的速度限制默认为 40MB/s，单位为字节。设置为 0 则不限速
indices.recovery.retry_delay_state_sync	由于集群状态同步导致 recovery 失败时，重试 recovery 前的等待时间，默认为 500 ms
indices.recovery.retry_delay_network	由于网络问题导致 recovery 失败时，重试 recovery 前的等待时间，默认为 5 s
indices.recovery.internal_action_timeout	用于某些恢复请求的 RPC 超时时间，默认为 15 min。 例如：prepare_translog、clean_files 等
indices.recovery.internal_action_long_timeout	与上面的用处相同，但是超时更长，默认为前者的 2 倍
indices.recovery.recovery_activity_timeout	不活跃的 recovery 超时时间，默认值等于 indices.recovery.internal_action_long_timeout

10.2 流程概述

recovery 由 clusterChanged 触发，从触发到开始执行恢复的调用关系如下：

```
indicesClusterStateService#applyClusterState
->createOrUpdateShards()
->createShard()
->indicesService.createShard()
->indexShard.startRecovery()
```

IndexShard#startRecovery 执行对一个特定分片的恢复流程，根据此分片不同的恢复类型执行相应的恢复过程：

```
public void startRecovery(...) {
    switch (recoveryState.getRecoverySource().getType()) {
```




```
case EMPTY_STORE:
case EXISTING_STORE:
    //主分片从本地恢复
    recoverFromStore();
    break;
case PEER:
    //副分片从远程主分片恢复
    recoveryTargetService.startRecovery(this,
recoveryState.getSourceNode(), recoveryListener);
    break;
case SNAPSHOT:
    //从快照恢复
    restoreFromRepository(repository);
    break;
case LOCAL_SHARDS:
    //从本节点的其他分片恢复 (shrink 时)
    recoverFromLocalShards();
    break;
default:
    throw new IllegalArgumentException("Unknown recovery source " +
recoveryState.getRecoverySource());
    }
}
```

此时执行线程池为 `clusterApplierService#updateTask`，执行具体的恢复工作时，会到另一个线程池中执行。无论哪种恢复类型，都在 `generic` 线程池中。

本章我们介绍主分片和副分片的恢复流程。`Snapshot` 和 `shrink` 属于比较独立的功能，在后续的章节中单独分析。

10.3 主分片恢复流程

1. INIT 阶段

一个分片的恢复流程中，从开始执行恢复的那一刻起，被标记为 `INIT` 阶段，`INIT` 标记在 `IndexShard#startRecovery` 函数的参数中传入，在判断此分片属于哪种恢复类型之前就被设置为 `INIT` 阶段。

```
markAsRecovering("from store", recoveryState);
```

然后新的线程池中执行主分片恢复流程:

```
threadPool.generic().execute(() -> {
    if (recoverFromStore()) {
        recoveryListener.onRecoveryDone(recoveryState);
    }
});
```

接下来, 恢复流程在新的线程池中开始执行, 开始阶段主要是一些验证工作, 例如, 校验当前分片是否为主分片, 分片状态是否异常等。

做完简单的校验工作后, 进入 INDEX 阶段:

```
public void prepareForIndexRecovery() {
    recoveryState.setStage(RecoveryState.Stage.INDEX);
}
```

2. INDEX 阶段

本阶段从 Lucene 读取最后一次提交的分段信息, 获取其中的版本号, 更新当前索引版本:

```
final Store store = indexShard.store();
si = store.readLastCommittedSegmentsInfo();
version = si.getVersion();
recoveryState.getIndex().updateVersion(version);
```

3. VERIFY_INDEX 阶段

VERIFY_INDEX 中的 INDEX 指 Lucene index, 因此本阶段的作用是验证当前分片是否损坏, 是否进行本项检查取决于配置项:

```
index.shard.check_on_startup
```

该配置的取值如下表所示。

阶 段	简 介
false	默认值, 打开分片时不检查分片是否损坏
checksum	检查物理损坏
true	检查物理和逻辑损坏, 这将消耗大量的内存和 CPU 资源

续表

阶 段	简 介
fix	检查物理和逻辑损坏。损坏的分段将被集群自动删除,这将导致数据丢失。使用时请考虑清楚

在索引的数据量较大时,分片检查会消耗更多的时间。

验证工作在 `IndexShard#checkIndex` 函数中完成。验证过程通过对比元信息中记录的 checksum 与 Lucene 文件的实际值,或者调用 Lucene `CheckIndex` 类中的 `checkIndex`、`exorciseIndex` 方法完成。

默认配置为不执行验证索引,进入最重要的 TRANSLOG 阶段。

4. TRANSLOG 阶段

一个 Lucene 索引由许多分段组成,每次搜索时遍历所有分段。内部维护了一个称为“提交点”的信息,其描述了当前 Lucene 索引都包括哪些分段,这些分段已经被 fsync 系统调用,从操作系统的 cache 刷入磁盘。每次提交操作都会将分段刷入磁盘实现持久化。

本阶段需要重放事务日志中尚未刷入磁盘的信息,因此,根据最后一次提交的信息做快照,来确定事务日志中哪些数据需要重放。重放完毕后将新生成的 Lucene 数据刷入磁盘。

```
private void recoverFromTranslogInternal() throws IOException {
    //根据最后一次提交的信息生成 translog 快照
    final long translogGen = Long.parseLong(lastCommittedSegmentInfos.
        getUserData().get(Translog.TRANSLOG_GENERATION_KEY));
    try (Translog.Snapshot snapshot = translog.newSnapshotFromGen
        (translogGen)) {
        //重放这些日志
        opsRecovered = config().getTranslogRecoveryRunner().run(this, snapshot);
    } catch (Exception e) {
        throw new EngineException(e);
    }
    //将重放后新生成的数据刷入硬盘
    if (opsRecovered > 0) {
        flush(true, true);
        refresh("translog_recovery");
    }
}
```

遍历所有需要重放的事务日志,执行具体的写操作,如同写入过程一样:

```

int runTranslogRecovery(Engine engine, Translog.Snapshot snapshot) throws
IOException {
    int opsRecovered = 0;
    Translog.Operation operation;
    while ((operation = snapshot.next()) != null) {
        //执行具体的写操作
        Engine.Result result = applyTranslogOperation();
        opsRecovered++;
    }
    return opsRecovered;
}

```

事务日志重放完毕后, StoreRecovery#internalRecoverFromStore 方法调用 indexShard.finalizeRecovery() 进入 FINALIZE 阶段。

5. FINALIZE 阶段

本阶段执行刷新 (refresh) 操作, 将缓冲的数据写入文件, 但不刷盘, 数据在操作系统的 cache 中。

```

public void finalizeRecovery() {
    recoveryState().setStage(RecoveryState.Stage.FINALIZE);
    Engine engine = getEngine();
    engine.refresh("recovery_finalization");
    engine.config().setEnableGcDeletes(true);
}

```

还是在 StoreRecovery#internalRecoverFromStore 方法中调用 indexShard.postRecovery, 将阶段设置为 DONE。

6. DONE 阶段

DONE 阶段是恢复工作的最后一个阶段, 进入 DONE 阶段之前再次执行 refresh, 然后更新分片状态。

```

public IndexShard postRecovery(String reason) throws IndexShardStartedException,
IndexShardRelocatedException, IndexShardClosedException {
    synchronized (mutex) {
        getEngine().refresh("post_recovery");
        recoveryState.setStage(RecoveryState.Stage.DONE);
    }
}

```

```

        changeState(IndexShardState.POST_RECOVERY, reason);
    }
    return this;
}

```

至此，主分片恢复完毕，对恢复结果进行处理。

如果恢复成功，则执行 `IndicesClusterStateService.RecoveryListener#onRecoveryDone`

主要实现是向 Master 发送 action 为 `internal:cluster/shard/started` 的 RPC 请求。

```
sendShardAction(SHARD_STARTED_ACTION_NAME, currentState, shardEntry, listener);
```

如果恢复失败，则执行 `IndicesClusterStateService#handleRecoveryFailure`。

主要实现是关闭 Engine，向 Master 发送 `internal:cluster/shard/failure` 的 RPC 请求。

```

private void failAndRemoveShard(...) {
    //关闭 Engine
    indexService.removeShard(shardRouting.shardId().id(), message);
    if (sendShardFailure) {
        //向 Master 发送 shard failure 消息
        sendFailShard(shardRouting, message, failure, state);
    }
}

```

10.4 副分片恢复流程

10.4.1 流程概述

副分片恢复的核心思想是从主分片拉取 Lucene 分段和 translog 进行恢复。按数据传递的方向，主分片节点称为 Source，副分片节点称为 Target。

为什么需要拉取主分片的 translog？因为在副分片恢复期间允许新的写操作，从复制 Lucene 分段的那一刻开始，所恢复的副分片数据不包括新增的内容，而这些内容存在于主分片的 translog 中，因此副分片需要从主分片节点拉取 translog 进行重放，以获取新增内容。这就需要主分片节点的 translog 不被清理。为了防止主分片节点的 translog 被清理，这方面的实现机制经历了多次迭代。

在 1.x 版本时代，通过阻止刷新（refresh）操作，让 translog 都保留下来。但是这样可能会产生很大的 translog。

在 2.0~5.x 版本时代，引入了 translog.view 的概念：

- 为了安全地完成 recoveries/relocations 操作，我们必须在 recovery 过程开始后保证所有的操作已全部处理（一个特定的索引或更新请求称为一个操作），以便重放。目前是通过防止 engine flush，从而确保操作 operations 都在 translog 中来实现的。这没问题，因为我们确实需要这些 operations。如果另一个 recovery 并发启动，则可能会有不必要的长时间重试。如果我们在这个时候因为某种原因关闭了 Engine（比如一个节点重新启动），当再次启动的时候，我们需要恢复一个很大的 translog。
- 为了解决这个问题，translog 被改为基于多个文件而不是一个文件。这允许 recovery 保留所需的文件，同时允许 Engine 执行 flush，以及执行 Lucene 的 commit（这将创建一个新的 translog 文件）。
- 重构了 translog 文件管理模块，允许存在多个文件。
- translog 维护一个引用文件的列表，包括未完成的 recovery，以及那些包含尚未提交到 Lucene 的 operations 的文件。
- 引入了新的 translog.view 概念，允许 recovery 获取一个引用，包括所有当前未提交的 translog 文件，以及所有未来新创建的 translog 文件，直到 view 关闭。它们可以使用这个 view 做 operations 的遍历操作。

创建一个视图可以获取后续的所有操作，直到关闭视图。刷新可以自由执行，translog 文件删除逻辑基于文件级别的引用计数。当获得一个视图时，这些计数器会增加。创建快照时同样会用到视图来读取 translog 中的操作，视图必须一直保存直到最后一个快照被消费完。这在 recovery 流程中没问题，但是在 Primary Replica Resyncer 中存在一个小 bug，只影响未发布的代码。具体请参考 <https://github.com/elastic/elasticsearch/pull/25862>。

因此从 6.0 版本开始，translog.view 被移除。引入 TranslogDeletionPolicy 的概念，负责维护活跃(liveness)的 translog 文件。这个类的实现非常简单，它将 translog 做一个快照来保持 translog 不被清理。这样使用者只需创建一个快照，无须担心视图之类。恢复流程实际上确实需要一个视图，现在可以通过获取一个简单的保留锁来防止清理 translog。这消除了视图概念的需求。

在保证 translog 不被清理后，恢复核心处理过程由两个内部阶段（phase）组成。

- phase1：在主分片所在节点，获取 translog 保留锁，从获取保留锁开始，会保留 translog 不受其刷盘清空的影响。然后调用 Lucene 接口把 shard 做快照，快照含有 shard 中已经刷到磁盘的文件引用，把这些 shard 数据复制到副本节点。在 phase1 结束前，会向

副分片节点发送告知对方启动 Engine，在 phase2 开始之前，副分片就可以正常处理写请求了。

- phase2: 对 translog 做快照，这个快照里包含从 phase1 开始，到执行 translog 快照期间的新增索引。将这些 translog 发送到副分片所在节点进行重放。

由于 phase1 需要通过网络复制大量数据，过程非常漫长，在 ES 6.x 中，有两个机会可以跳过 phase1:

- (1) 如果可以基于恢复请求中的 SequenceNumber 进行恢复，则跳过 phase1。
- (2) 如果主副两分片有相同的 syncid 且 doc 数相同，则跳过 phase1。

在数据模型一章中介绍过 SequenceNumber，现在介绍 syncid 的概念。

10.4.2 synced flush 机制

为了解决副分片恢复过程第一阶段时间太漫长而引入了 synced flush，默认情况下 5 分钟没有写入操作的索引被标记为 inactive，执行 synced flush，生成一个唯一的 syncid，写入分片的所有副本中。这个 syncid 是分片级，意味着拥有相同 syncid 的分片具有相同的 Lucene 索引。

synced flush 本质上是一次普通的 flush 操作，只是在 Lucene 的 commit 过程中多写了一个 syncid。原则上，在没有数据写入的情况下，各分片在同一时间“flush”成功后，它们理应有相同的 Lucene 索引内容，无论 Lucene 分段是否一致。于是给分片分配一个 id，表示数据一致。

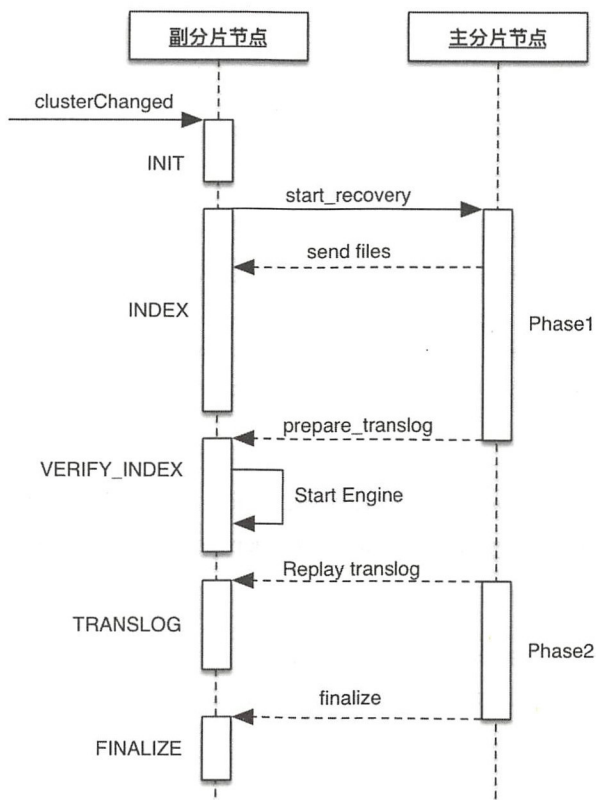
但是显然 synced flush 期间不能有新写入的内容，如果 syncflush 执行期间收到写请求，则 ES 选择了写入可用性：让 synced flush 失败，让写操作成功。在没有执行 flush 的情况下已有 syncid 不会失效。

在某个分片上执行普通 flush 操作会删除已有 syncid。因此，synced flush 操作是一个不可靠操作，只适用于冷索引。

下面详细介绍整个副分片恢复过程。

10.4.3 副分片节点处理过程

副分片恢复的 VERIFY_INDEX、TRANSLOG、FINALIZE 三个阶段由主分片节点发送的 RPC 调用触发，如下图所示。



在副分片恢复过程中，副分片节点会向主分片节点发送 `start_recovery` 的 RPC 请求，主分片节点对此请求的处理注册在 `PeerRecoverySourceService` 类中，内部类 `PeerRecoverySourceService.StartRecoveryTransportRequestHandler` 负责处理此 RPC 请求。

类似的，主分片节点也会向副分片节点发送一些 RPC 请求，副分片节点对这些请求的处理以 `XXXRequestHandler` 的方式注册在 `PeerRecoveryTargetService` 类中，包括接收 Lucene 文件、接收 translog 并重放、执行清理等操作，如下图所示。



1. INIT 阶段

本阶段在副分片节点执行。

与主分片恢复的 INIT 阶段类似, 恢复任务开始时被设置为 INIT 阶段, 进行副分片恢复时, 在新的线程池中执行恢复任务:

```
public void startRecovery(...) {  
    final long recoveryId = onGoingRecoveries.startRecovery(indexShard,  
sourceNode, listener,  
        recoverySettings.activityTimeout());  
    threadPool.generic().execute(new RecoveryRunner(recoveryId));  
}
```

构建准备发往主分片的 StartRecoveryRequest 请求, 请求中包括将本次要恢复的 shard 相关信息, 如 shardid、metadataSnapshot 等。metadataSnapshot 中包含 syncid。

然后进入 INDEX 阶段:

```
public void prepareForIndexRecovery() {  
    recoveryState.setStage(RecoveryState.Stage.INDEX);  
}
```

2. INDEX 阶段

INDEX 阶段负责将主分片的 Lucene 数据复制到副分片节点。

向主分片节点发送 action 为 internal:index/shard/recovery/start_recovery 的 PRC 请求, 并阻塞当前线程, 等待响应, 直到对方处理完成。然后设置为 DONE 阶段。

概括来说, 主分片节点收到请求后把 Lucene 和 translog 发送给副分片, 具体参考主分片节点处理过程一节。下面是副分片节点发送请求并等待响应的过程:

```
private void doRecovery(final long recoveryId) {  
    final AtomicReference<RecoveryResponse> responseHolder = new  
AtomicReference<>();  
    cancellableThreads.execute(() -> responseHolder.set(  
        transportService.submitRequest(request.sourceNode(),  
PeerRecoverySourceService.Actions.START_RECOVERY, request,  
            new FutureTransportResponseHandler<RecoveryResponse>()).  
txGet())); //txGet 阻塞线程等待响应  
    //设置为 DONE 阶段
```



```
onGoingRecoveries.markRecoveryAsDone(recoveryId);
}
```

线程阻塞等待 INDEX 阶段完成，然后直接到 DONE 阶段。在这期间主分片节点会发送几次 RPC 调用，通知副分片节点启动 Engine，执行清理等操作。VERIFY_INDEX 和 TRANSLOG 阶段也是由主分片节点的 RPC 调用触发的。

3. VERIFY_INDEX 阶段

副分片的索引验证过程与主分片相同，是否进行验证取决于配置。默认为不执行索引验证。

主分片节点执行完 phase1 后，调用 prepareTargetForTranslog 方法，向副分片节点发送 action 为 internal:index/shard/recovery/prepare_translog 的 RPC 请求。副分片对此 action 的主要处理是启动 Engine，使副分片可以正常接收写请求。副分片的 VERIFY_INDEX、TRANSLOG 两阶段也是在对这个 action 的处理中触发的。调用链如下：

```
recoveryRef.target().prepareForTranslogOperations()
RecoveryTarget#prepareForTranslogOperations()
IndexShard#skipTranslogRecovery()
IndexShard#internalPerformTranslogRecovery
```

主分片和副分片的 VERIFY_INDEX、TRANSLOG 都在 IndexShard#internalPerformTranslogRecovery 方法中实现。

```
private void internalPerformTranslogRecovery(boolean skipTranslogRecovery,
boolean indexExists) throws IOException {
    //进入 VERIFY_INDEX 阶段
    recoveryState.setStage(RecoveryState.Stage.VERIFY_INDEX);
    //默认为不执行索引验证
    if (Booleans.isTrue(checkIndexOnStartup)) {
        checkIndex();
    }
    //进入 TRANSLOG 阶段
    recoveryState.setStage(RecoveryState.Stage.TRANSLOG);
    if (indexExists == false) {
        openMode = EngineConfig.OpenMode.CREATE_INDEX_AND_TRANSLOG;
    } else if (skipTranslogRecovery) {
        //副分片恢复执行到此处，跳过后面的 newEngine.recoverFromTranslog
        openMode = EngineConfig.OpenMode.OPEN_INDEX_CREATE_TRANSLOG;
```



```
    } else {  
        //主分片恢复执行到此处, 创建 Engine 后执行 ewEngine.recoverFromTranslog  
        openMode = EngineConfig.OpenMode.OPEN_INDEX_AND_TRANSLOG;  
    }  
  
    //创建新的 Engine  
    final EngineConfig config = newEngineConfig(openMode);  
    Engine newEngine = createNewEngine(config);  
    if (openMode == EngineConfig.OpenMode.OPEN_INDEX_AND_TRANSLOG) {  
        //主分片的 translog 恢复流程  
        newEngine.recoverFromTranslog();  
    }  
}
```

4. TRANSLOG 阶段

TRANSLOG 阶段负责将主分片的 translog 数据复制到副分片节点进行重放。

先创建新的 Engine, 跳过 Engine 自身的 translog 恢复。此时主分片的 phase2 尚未开始, 接下来的 TRANSLOG 阶段就是等待主分片节点将 translog 发到副分片节点进行重放, 也就是 phase2 的执行过程。

5. FINALIZE 阶段

主分片节点执行完 phase2, 调用 finalizeRecovery, 向副分片节点发送 action 为 internal:index/shard/recovery/finalize 的 RPC 请求, 副分片节点对此 action 的处理为先更新全局检查点, 然后执行与主分片相同的清理操作:

```
public void finalizeRecovery(final long globalCheckpoint) {  
  
    //更新全局检查点, 尚未进入 FINALIZE 阶段  
    indexShard().updateGlobalCheckpointOnReplica(globalCheckpoint,  
    "finalizing recovery");  
    final IndexShard indexShard = indexShard();  
    //进入 FINALIZE 阶段, 执行与主分片相同的清理操作, 主要是 refresh  
    indexShard.finalizeRecovery();  
}
```

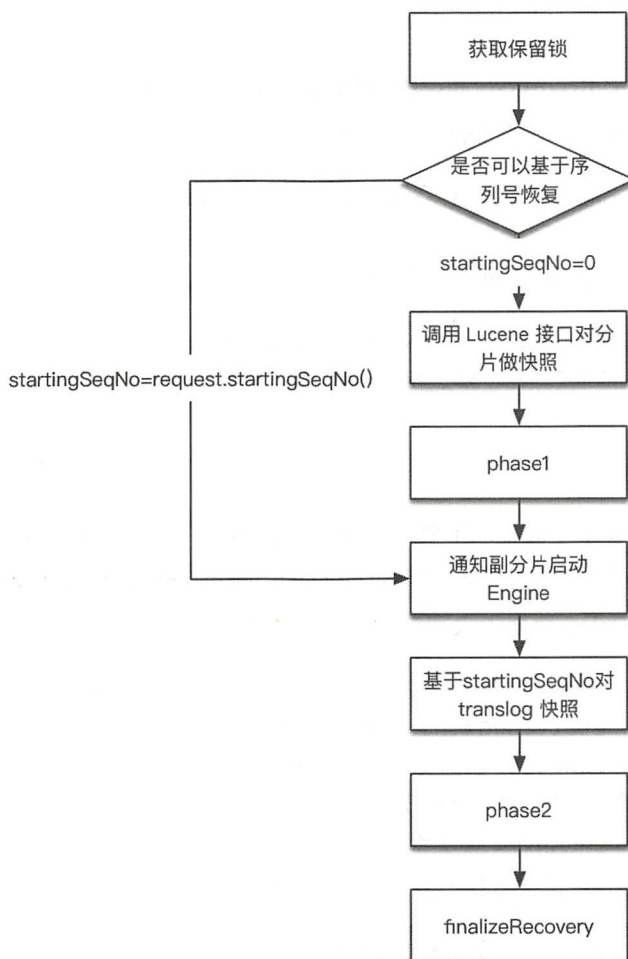
6. DONE 阶段

副分片节点等待 INDEX 阶段执行完成后, 调用 onGoingRecoveries.markRecoveryAsDone

(recoveryId) 进入 DONE 阶段。主要处理是调用 `indexShard#postRecovery`，与主分片的 `postRecovery` 处理过程相同，包括对恢复成功或失败的处理，也和主分片的处理过程相同。

10.4.4 主分片节点处理过程

副分片恢复的 INDEX 阶段向主分片节点发送 action 为 `internal:index/shard/recovery/start_recovery` 的恢复请求，主分片对此请求的处理过程是副分片恢复的核心流程。核心流程如下图所示。



整体处理流程

主分片节点收到副分片节点发送的恢复请求，执行恢复，然后返回结果，这里也是阻塞处

理的过程，下面的消息处理在 generic 线程池中执行。

```
class StartRecoveryTransportRequestHandler implements TransportRequestHandler
<StartRecoveryRequest> {
    public void messageReceived(final StartRecoveryRequest request, final
TransportChannel channel) throws Exception {
        RecoveryResponse response = recover(request);
        channel.sendResponse(response);
    }
}
```

主要处理流程位于 RecoverySourceHandler#recoverToTarget。

首先获取一个保留锁，使得 translog 不被清理：

```
Closeable ignored = shard.acquireTranslogRetentionLock()
```

判断是否可以从 SequenceNumber 恢复：

```
final boolean isSequenceNumberBasedRecoveryPossible = request.startingSeqNo() !=
SequenceNumbers.UNASSIGNED_SEQ_NO &&
    isTargetSameHistory() && isTranslogReadyForSequenceNumber-
BasedRecovery();
```

除了异常检测和版本号检测，主要在 isTranslogReadyForSequenceNumberBasedRecovery 方法中判断请求的序列号是否小于主分片节点的 localCheckpoint，以及 translog 中的数据是否足以恢复（有可能因为 translog 数据太大或过期删除而无法恢复）。

```
boolean isTranslogReadyForSequenceNumberBasedRecovery() throws IOException {
    final long startingSeqNo = request.startingSeqNo();
    final long localCheckpoint = shard.getLocalCheckpoint();
    //正常情况下应该都小于主分片的本地检查点
    if (startingSeqNo - 1 <= localCheckpoint) {
        final LocalCheckpointTracker tracker = new LocalCheckpointTracker
(startingSeqNo, startingSeqNo - 1);
        //检查 translog 中的数据是否足够
        try (Translog.Snapshot snapshot = shard.getTranslog().newSnapshot-
FromMinSeqNo(startingSeqNo)) {
            Translog.Operation operation;
```

```
        while ((operation = snapshot.next()) != null) {
            if (operation.seqNo() != SequenceNumbers.UNASSIGNED_SEQ_NO) {
                tracker.markSeqNoAsCompleted(operation.seqNo());
            }
        }
    }
    return tracker.getCheckpoint() >= localCheckpoint;
} else {
    return false;
}
}
```

以请求的序列号作为最小值做一个快照，遍历这个值从开始到最新的数据之间的操作，检查序列号验证事务日志中的操作是否完整。

如果可以基于 `SequenceNumber` 恢复，则跳过 `phase1`，否则调用 `Lucene` 接口对分片做快照，执行 `phase1`。

```
phase1Snapshot = shard.acquireIndexCommit(false);
phase1(phase1Snapshot.getIndexCommit(), translog::totalOperations);
```

`phase1` 的执行过程我们单独论述。

等待 `phase1` 执行完毕，主分片节点通知副分片节点启动此分片的 `Engine`：

```
prepareTargetForTranslog(translog.estimateTotalOperationsFromMinSeq(startingSeqNo));
```

该方法会阻塞处理，直到分片 `Engine` 启动完毕。待副分片启动 `Engine` 完毕，就可以正常接收写请求了。注意，此时 `phase2` 尚未开始，此分片的恢复流程尚未结束。

等待当前操作处理完成后，以 `startingSeqNo` 为起始点，对 `translog` 做快照，开始执行 `phase2`：

```
try(Translog.Snapshot snapshot = translog.newSnapshotFromMinSeqNo(
    startingSeqNo)) {
    targetLocalCheckpoint = phase2(startingSeqNo, requiredSeqNoRangeStart,
    endingSeqNo, snapshot);
} catch (Exception e) {
    throw new RecoveryEngineException(shard.shardId(), 2, "phase2 failed", e);
}
```

如果基于 SequenceNumber 恢复, 则 startingSeqNo 取值为恢复请求中的序列号, 从请求的序列号开始快照 translog。否则取值为 0, 快照完整的 translog。

最后调用 RecoverySourceHandler#finalizeRecovery 执行清理工作, 该方法向副分片节点发送 action 为 internal:index/shard/recovery/finalize 的 RPC 请求告知对方执行清理, 同时把全局检查点发送过去, 等待对方执行成功, 主分片更新全局检查点。

phase1

phase1 检查目标节点上的段文件, 并对缺失的部分进行复制。只有具有相同大小和校验和的段才能被重用。但是由于分片副本执行各自的合并策略, 所以合并出来的段文件相同的概率很低。

phase1 的实现位于 RecoverySourceHandler#phase1。

在对比分段之前, 先检查主副两分片是否都有 syncid, 如果 syncid 相同, 且 doc 数相同, 则跳过 phase1。

```
String recoverySourceSyncId = recoverySourceMetadata.getSyncId();
String recoveryTargetSyncId = request.metadataSnapshot().getSyncId();
final boolean recoverWithSyncId = recoverySourceSyncId != null &&
    recoverySourceSyncId.equals(recoveryTargetSyncId);
if (recoverWithSyncId) {
    final long numDocsTarget = request.metadataSnapshot().getNumDocs();
    final long numDocsSource = recoverySourceMetadata.getNumDocs();
    if (numDocsTarget != numDocsSource) {
        throw new IllegalStateException();
    }
}
```

否则对比文件差异, 发送文件:

```
final Store.RecoveryDiff diff = recoverySourceMetadata.recoveryDiff
(request.metadataSnapshot());
List<StoreFileMetadata> phase1Files = new ArrayList<>(diff.different.size()
+ diff.missing.size());
phase1Files.addAll(diff.different);
phase1Files.addAll(diff.missing);
sendFiles(store, phase1Files.toArray(new StoreFileMetadata[phase1Files.size()]),
outputStreamFactories);
```


phase2

phase2 将 translog 批量发送到副分片节点, 发送时将待发送的 translog 组合成一批来提高发送效率, 默认的批量大小为 512KB, 不支持配置。主要实现如下:

```
protected SendSnapshotResult sendSnapshot() {
    Translog.Operation operation;
    //遍历快照中的操作
    while ((operation = snapshot.next()) != null) {
        operations.add(operation);
        size += operation.estimateSize();
        totalSentOps++;

        //准备一批数据
        if (size >= chunkSizeInBytes) {
            cancellableThreads.executeIO(sendBatch);
            size = 0;
            operations.clear();
        }
    }
    if (!operations.isEmpty() || totalSentOps == 0) {
        //发送到副分片节点
        cancellableThreads.executeIO(sendBatch);
    }
}
```

10.5 recovery 速度优化

众所周知, 索引恢复是集群启动过程中最缓慢的过程, 集群完全重启, 或者 Master 节点挂掉后, 新选出的 Master 也有可能执行这个过程。

官方也一直在优化索引恢复速度, 陆续添加了 syncid 和 SequenceNumber。下面归纳一下有哪些方法可以提升索引恢复速度:

- 配置项 `cluster.routing.allocation.node_concurrent_recoveries` 决定了单个节点执行副分片 recovery 时的最大并发数(进/出), 默认为 2, 适当提高此值可以增加 recovery 并发数。
- 配置项 `indices.recovery.max_bytes_per_sec` 决定节点间复制数据时的限速, 可以适当提高此值或取消限速。

- 配置项 `cluster.routing.allocation.node_initial_primaries_recoveries` 决定了单个节点执行主分片 `recovery` 时的最大并发数, 默认为 4。由于主分片的恢复不涉及在网络上复制数据, 仅在本地的磁盘读写, 所以在节点配置了多个数据磁盘的情况下, 可以适当提高此值。
- 在重启集群之前, 先停止写入端, 执行 `sync flush`, 让恢复过程有机会跳过 `phase1`。
- 适当地多保留些 `translog`, 配置项 `index.translog.retention.size` 默认最大保留 512MB, `index.translog.retention.age` 默认为不超过 12 小时。调整这两个配置可让恢复过程有机会跳过 `phase1`。
- 合并 Lucene 分段, 对于冷索引甚至不再更新的索引执行 `_forcemerge`, 较少的 Lucene 分段可以提升恢复效率, 例如, 减少对比, 降低文件传输请求数量。

最后, 可以考虑允许主副分片存在一定程度的不一致, 修改 ES 恢复流程, 少量的不一致则跳过 `phase1`。

10.6 如何保证副分片和主分片一致

索引恢复过程的一个难点在于如何维护主副分片的一致性。假设副分片恢复期间一直有写操作, 如何实现一致呢?

我们先看看早期的做法: 在 2.0 版本之前, 副分片恢复要经历三个阶段。

- `phase1`: 将主分片的 Lucene 做快照, 发送到 `target`。期间不阻塞索引操作, 新增数据写到主分片的 `translog`。
- `phase2`: 将主分片 `translog` 做快照, 发送到 `target` 重放, 期间不阻塞索引操作。
- `phase3`: 为主分片加写锁, 将剩余的 `translog` 发送到 `target`。此时数据量很小, 写入过程的阻塞很短。

从理论上来说, 只要流程上允许将写操作阻塞一段时间, 实现主副一致是比较容易的。

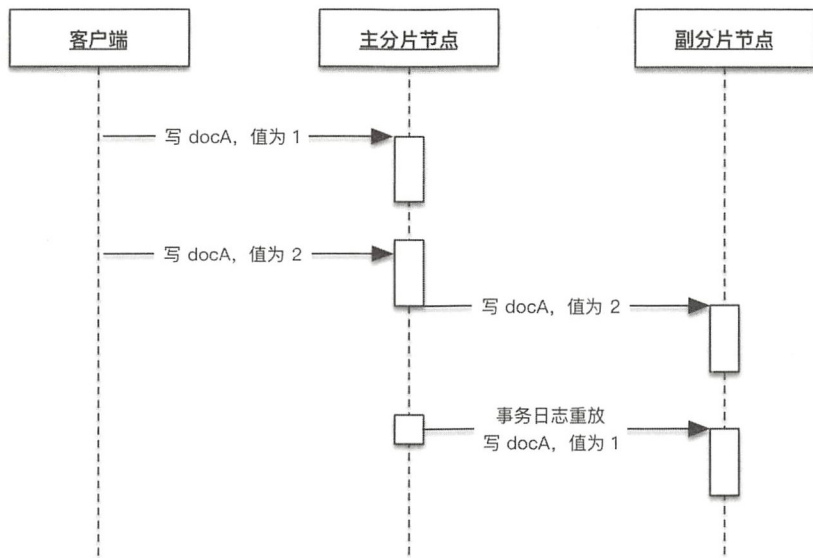
但是后来 (从 2.0 版本开始), 也就是引入 `translog.view` 概念的同时, `phase3` 被删除。

`phase3` 被删除, 这个阶段是重放操作 (`operations`), 同时防止新的写入 Engine。这是不必要的, 因为自恢复开始, 标准的 `index` 操作会发送所有的操作到正在恢复中的分片。重放恢复开始时获取的 `view` 中的所有操作足够保证不丢失任何操作。

阻塞写操作的 `phase3` 被删除, 恢复期间没有任何写阻塞过程。接下来需要处理的就是解决 `phase1` 和 `phase2` 之间的写操作与 `phase2` 重放操作之间的时序和冲突问题。在副分片节点, `phase1` 结束后, 假如新增索引操作和 `translog` 重放操作并发执行, 因为时序的关系会出现新老数据交替。如何实现主副分片一致呢?

假设在第一阶段执行期间, 有客户端索引操作要求将 `docA` 的内容写为 1, 主分片执行了这

个操作，而副分片由于尚未就绪所以没有执行。第二阶段期间客户端索引操作要求写 docA 的内容为 2，此时副分片已经就绪，先执行将 docA 写为 2 的新增请求，然后又收到了从主分片所在节点发送过来的 translog 重复写 docA 为 1 的请求该如何处理？具体流程如下图所示。



答案是在写流程中做异常处理，通过版本号来过滤掉过期操作。写操作有三种类型：索引新文档、更新、删除。索引新文档不存在冲突问题，更新和删除操作采用相同的处理机制。每个操作都有一个版本号，这个版本号就是预期 doc 版本，它必须大于当前 Lucene 中的 doc 版本号，否则就放弃本次操作。对于更新操作来说，预期版本号是 Lucene doc 版本号+1。主分片节点写成功后新数据的版本号会放到写副本的请求中，这个请求中的版本号就是预期版本号。

这样，时序上存在错误的操作被忽略，对于特定 doc，只有最新一次操作生效，保证了主副分片一致。

我们分别看一下写操作三种类型的处理机制。

1. 索引新文档

不存在冲突问题，不需要处理。

2. 更新

判断本次操作的版本号是否小于 Lucene 中 doc 的版本号，如果小于，则放弃本次操作。

Index、Delete 都继承自 Operation，每个 Operation 都有一个版本号，这个版本号就是 doc 版本号。对于副分片的写流程来说，正常情况下是主分片写成功后，相应 doc 写入的版本号被放到转发写副分片的请求中。对于更新来说，就是通过主分片将原 doc 版本号+1 后转发到副分

片实现的。在对比版本号的时候：

expectedVersion = 写副分片请求中的 version = 写主分片成功后的 version

通过下面的方法判断当前操作的版本号是否低于 Lucene 中的版本号：

```
EXTERNAL((byte) 1) {
    @Override
    public boolean isVersionConflictForWrites(long currentVersion, long
expectedVersion, boolean deleted) {
        if (currentVersion == Versions.NOT_FOUND) {
            return false;
        }
        if (expectedVersion == Versions.MATCH_ANY) {
            return true;
        }
        if (currentVersion >= expectedVersion) {
            return true;
        }
        return false;
    }
}
```

如果 translog 重放的操作在写一条“老”数据，则 compareOpToLuceneDocBasedOnVersions 会返回 OpVsLuceneDocStatus.OP_STALE_OR_EQUAL。

```
private OpVsLuceneDocStatus compareOpToLuceneDocBasedOnVersions(final
Operation op)
    throws IOException {
    final VersionValue versionValue = resolveDocVersion(op);
    if (versionValue == null) {
        return OpVsLuceneDocStatus.LUCENE_DOC_NOT_FOUND;
    } else {
        return op.versionType().isVersionConflictForWrites(versionValue.
version, op.version(), versionValue.isDelete()) ?
            OpVsLuceneDocStatus.OP_STALE_OR_EQUAL : OpVsLuceneDocStatus.OP_NEWER;
    }
}
```

副分片在 InternalEngine#index 函数中通过 plan 判断是否写到 Lucene：

```
// non-primary mode (i.e., replica or recovery)
plan = planIndexingAsNonPrimary(index);
```

在 `planIndexingAsNonPrimary` 函数中，`plan` 的最终结果就是 `plan = IndexingStrategy.processButSkipLucene`，后面会跳过写 Lucene 和 translog 的逻辑。

3. 删除

判断本次操作中的版本号是否小于 Lucene 中 doc 的版本号，如果小于，则放弃本次操作。

通过 `compareOpToLuceneDocBasedOnVersions` 方法判断本次操作是否小于 Lucene 中 doc 的版本号，与 Index 操作时使用相同的比较函数。

类似的，在 `InternalEngine#delete` 函数中判断是否写到 Lucene：

```
plan = planDeletionAsNonPrimary(delete);
```

如果 translog 重放的是一个“老”的删除操作，则 `compareOpToLuceneDocBasedOnVersions` 会返回 `OpVsLuceneDocStatus.OP_STALE_OR_EQUAL`。

`plan` 的最终结果就是 `plan=DeletionStrategy.processButSkipLucene`，后面会跳过 Lucene 删除的逻辑。

10.7 recovery 相关监控命令

在实际生产环境中我们经常需要了解 recovery 的进度和状态，ES 提供了丰富的 API 可以获取这些信息。

1. _cat/recovery

列出活跃的和已完成的 recovery 信息：

```
curl localhost:9200/_cat/recovery
website 0 30ms peer done 127.0.0.1 fc6s0S0 127.0.0.1 iV6xmvW n/a
n/a 0 0 0.0% 0 0 0 0.0% 0 0 0 100.0%
website 0 30ms existing_store done n/a n/a 127.0.0.1 fc6s0S0 n/a
n/a 0 0 100.0% 14 0 0 100.0% 13825 0 0 100.0%
```

使用 `?v` 参数可以显示列名：

```
GET _cat/recovery?v
```


这个 API 提供的信息包括 `recovery` 类型，`existing_store` 意味着主分片本地恢复，`peer` 代表副分片从其他节点恢复，以及数据传输的源节点和目的节点信息，数据传输进度，总体文件大小和已传输字节数，等等。

此 API 的完整的参数列表请参考手册地址为 <https://www.elastic.co/guide/en/elasticsearch/reference/current/cat-recovery.html>。

2. {index}/_recovery

此 API 展示特定索引的 `recovery` 所处阶段，以及每个分片、每个阶段的详细信息。以索引“`website`”为例，返回信息摘要如下：

```
curl "localhost:9200/website/_recovery?pretty"
```

```
{
  "id" : 4,
  "type" : "PEER",
  "stage" : "DONE",
  "primary" : false,
  ...
  "source" : {
    ...
  },
  "target" : {
    ...
  },
  "index" : {
    "size" : {
      "total_in_bytes" : 0,
      "reused_in_bytes" : 0,
      "recovered_in_bytes" : 0,
      "percent" : "0.0%"
    }
    "total_time_in_millis" : 6,
    "source_throttle_time_in_millis" : 0,
    "target_throttle_time_in_millis" : 0
  },
  "translog" : {
    "recovered" : 0,
```

```

    "total" : 0,
    "percent" : "100.0%",
    "total_on_start" : -1,
    "total_time_in_millis" : 13
  },
  "verify_index" : {
    "check_index_time_in_millis" : 0,
    "total_time_in_millis" : 0
  }
},

```

除了基础信息，此 API 还给出了 `index`、`translog`、`verify_index` 三个耗时最长的重要阶段的详细信息。例如，传输字节数、限速信息等。如果更关注进度信息，则可以过滤 `stage` 和 `percent` 字段。

3. `_stats`

有时我们想知道 `sync flush` 是否完成，`_stats` API 可以给出分片级信息，包括分片的 `sync_id`、`local_checkpoint`、`global_checkpoint` 等，可以通过指定索引名称实现，或者使用 `_all` 输出全部索引的信息。

```

curl -s "127.0.0.1:9200/_all/_stats?level=shards&pretty" |grep sync_id
|wc -l

```

这个示例统计集群所有分片中有多少个分片拥有 `sync_id`。这意味着 `sync flush` 成功。

10.8 小结

- 主分片恢复的主要阶段是 `TRANSLOG` 阶段；
- 副分片恢复的主要阶段是 `INDEX` 和 `TRANSLOG` 阶段；
- 只有 `phase1` 有限速配置，`phase2` 不限速；
- Lucene 的“提交”概念就是从操作系统内存 `cache fsync` 到磁盘的过程。

11 chapter

第 11 章 gateway 模块分析

gateway 模块负责集群元信息的存储和集群重启时的恢复。

11.1 元数据

ES 中存储的数据有以下几种：

- state 元数据信息；
- index Lucene 生成的索引文件；
- translog 事务日志。

元数据信息又有以下几种：

- nodes/0/_state/*.st, 集群层面元信息；
- nodes/0/indices/{index_uuid}/_state/*.st, 索引层面元信息；
- nodes/0/indices/{index_uuid}/0/_state/*.st, 分片层面元信息。

分别对应 ES 中的数据结构：

- MetaData（集群层），主要是 clusterUUID、settings、templates 等；
- IndexMetaData（索引层），主要是 numberOfShards、mappings 等；
- ShardStateMetaData（分片层），主要是 version、indexUUID、primary 等。

上述信息被持久化到磁盘，需要注意的是：持久化的 state 不包括某个分片存在于哪个节点

这种内容路由信息，集群完全重启时，依靠 gateway 的 recovery 过程重建 RoutingTable。当读取某个文档时，根据路由算法确定目的分片后，从 RoutingTable 中查找分片位于哪个节点，然后将请求转发到目的节点。

11.2 元数据的持久化

只有具备 Master 资格的节点和数据节点可以持久化集群状态。当收到主节点发布的集群状态时，节点判断元信息是否发生变化，如果发生变化，则将其持久化到磁盘中。

GatewayMetaState 类负责接收集群状态，它继承自 ClusterStateApplier，并实现其中的 applyClusterState 方法，当收到新的集群状态时，ClusterApplierService 通知全部 applier 应用该集群状态：

```
private void callClusterStateAppliers(ClusterChangedEvent clusterChangedEvent) {
    //遍历全部的 applier，依次调用各模块对集群状态的处理
    clusterStateAppliers.forEach(applier -> {
        try {
            //调用各模块实现的 applyClusterState
            applier.applyClusterState(clusterChangedEvent);
        } catch (Exception ex) {
            //某个模块应用集群状态出现异常时打印日志，但应用过程不会终止
            logger.warn("failed to notify ClusterStateApplier", ex);
        }
    });
}
```

集群状态的发布、应用的详细过程请参考 Cluster 模块分析一章。

节点校验本身的资格，判断元信息是否发生变化，并将其持久化到磁盘中，全局元信息和索引级元信息都来自集群状态。

```
public void applyClusterState(ClusterChangedEvent event) {
    //只有具备 Master 资格的节点和数据节点才会持久化元数据
    if (state.nodes().getLocalNode().isMasterNode() || state.nodes().
        getLocalNode().isDataNode()) {
        // 检查全局元信息是否发生变化，如果有变化，则将其写入磁盘
        if (previousMetaData == null || !MetaData.isGlobalStateEquals
            (previousMetaData, newMetaData)) {
            metaStateService.writeGlobalState("changed", newMetaData);
        }
    }
}
```

```

    }

    //将发生变化的索引级元信息写入磁盘
    for (IndexMetaWriteInfo indexMetaWrite : writeInfo) {
        metaStateService.writeIndex(indexMetaWrite.reason, indexMetaWrite.
newMetaData);
    }
}
}
}

```

执行文件写入的过程封装在 `MetaDataStateFormat` 类中，全局元信息和索引级元信息的写入都执行三个流程：写临时文件、刷盘、“move”成目标文件。

```

//临时文件名
final Path tmpStatePath = stateLocation.resolve(fileName + ".tmp");
//目标文件名
final Path finalStatePath = stateLocation.resolve(fileName);

//写临时文件
OutputStreamIndexOutput out = new OutputStreamIndexOutput(Files.
newOutputStream(tmpStatePath),...);
out.writeInt(format.index());

//从系统 cache 刷到磁盘中，保证持久化
IOUtils.fsync(tmpStatePath, false); // fsync the state file
//move 为目标文件，move 操作为系统原子操作
Files.move(tmpStatePath, finalStatePath, StandardCopyOption.ATOMIC_MOVE);

```

11.3 元数据的恢复

上述的三种元数据信息被持久化存储到集群的每个节点，当集群完全重启（full restart）时，由于分布式系统的复杂性，各个节点保存的元数据信息可能不同。此时需要选择正确的元数据作为权威元数据。

gateway 的 recovery 负责找到正确的元数据，应用到集群。

当集群完全重启，达到 recovery 条件时，进入元数据恢复流程，一般情况下，recovery 条件由以下三个配置控制。

- `gateway.expected_nodes`, 预期的节点数。加入集群的节点数(数据节点或具备 Master 资格的节点) 达到这个数量后立即开始 gateway 的恢复。默认为 0。
- `gateway.recover_after_time`, 如果没有达到预期的节点数量, 则恢复过程将等待配置的时间, 再尝试恢复。默认为 5min。
- `gateway.recover_after_nodes`, 只要配置数量的节点(数据节点或具备 Master 资格的节点) 加入集群就可以开始恢复。

假设取值为 10、5min、8, 则集群启动时节点达到 10 个则立即进入 recovery; 如果一直没有达到 10 个, 5min 超时后如果节点达到 8 个也进入 recovery。

还有一些更细致的配置项, 原理与上面三个配置类似:

- `gateway.expected_master_nodes`, 预期的具备 Master 资格的节点数, 加入集群的具备 Master 资格的节点数达到这个数量后立即开始 gateway 的恢复。默认为 0。
- `gateway.expected_data_nodes`, 预期的具备数据节点资格的节点数, 加入集群的具备数据节点资格的节点数量达到这个数量后立即开始 gateway 的恢复。默认为 0。
- `gateway.recover_after_master_nodes`, 指定数量的具备 Master 资格的节点加入集群后就可以开始恢复。
- `gateway.recover_after_data_nodes`, 指定数量的数据节点加入集群后就可以开始恢复。

当集群完全启动时, gateway 模块负责集群层和索引层的元数据恢复, 分片层的元数据恢复在 allocation 模块实现, 但是由 gateway 模块在执行完上述两个层次恢复工作后触发。

当集群级、索引级元数据选举完毕后, 执行 `submitStateUpdateTask` 提交一个 `source` 为 `local-gateway-elected-state` 的任务, 触发获取 shard 级元数据的操作, 这个 Fetch 过程是异步的, 根据集群分片数量规模, Fetch 过程可能比较长, 然后 submit 任务就结束, gateway 流程结束。

因此, 三个层次的元数据恢复是由 gateway 模块和 allocation 模块共同完成的, 在 Gateway 将集群级、索引级元数据选举完毕后, 在 `submitStateUpdateTask` 提交的任务中会执行 allocation 模块的 `reroute` 继续后面的流程。

11.4 元数据恢复流程分析

主要实现在 `GatewayService` 类中, 它继承自 `ClusterStateListener`, 在集群状态发生变化 (`clusterChanged`) 时触发, 仅由 Master 节点执行。

(1) Master 选举成功之后, 判断其持有的集群状态中是否存在 `STATE_NOT_RECOVERED_BLOCK`, 如果不存在, 则说明元数据已经恢复, 跳过 gateway 恢复过程, 否则等待。



(2) Master 从各个节点主动获取元数据信息。

(3) 从获取的元数据信息中选择版本号最大的作为最新元数据，包括集群级、索引级。

(4) 两者确定之后，调用 allocation 模块的 reroute，对未分配的分片执行分配，主分片分配过程中会异步获取各个 shard 级别元数据，默认超时为 13s。

集群级和索引级元数据信息是根据存储在其中的版本号来选举的，而主分片位于哪个节点却是 allocation 模块动态计算出来的，先前主分片不一定还被选为新的主分片。关于主分片选举策略我们在 allocation 一章中介绍。

11.4.1 选举集群级和索引级别的元数据

判断是否满足进入 recovery 条件：实现位于 GatewayService#clusterChanged，执行此流程的线程为 clusterApplierService#updateTask。

此处省略相关的代码引用。当满足条件时，进入 recovery 主要流程：实现位于 Gateway#performStateRecovery；执行此流程的线程为 generic。

首先向有 Master 资格的节点发起请求，获取它们存储的元数据：

```
//具有 Master 资格的节点列表
String[] nodesIds = clusterService.state().nodes().getMasterNodes().
keys().toArray(String.class);
//发送获取请求并等待结果
TransportNodesListGatewayMetaState.NodesGatewayMetaState nodesState =
listGatewayMetaState.list(nodesIds, null).actionGet();
```

等待回复时，必须收到所有节点的回复，无论回复成功还是失败（节点通信失败异常会被捕获，作为失败处理），此处没有超时。

在收齐的这些回复中，有效元信息的总数必须达到指定数量。异常情况下，例如，某个节点上元信息读取失败，则回复信息中元数据为空。

```
int requiredAllocation = Math.max(1, minimumMasterNodesProvider.get());
```

minimumMasterNodesProvider 的值由下面的配置项决定：

```
discovery.zen.minimum_master_nodes
```

接下来就是通过版本号选取集群级和索引级元数据。



选举集群级元数据:

```
public void performStateRecovery(...)
{
    //遍历请求的所有节点
    for (TransportNodesListGatewayMetaState.NodeGatewayMetaState
nodeState : nodesState.getNodes()) {
        //根据元信息中记录的版本号选举元信息
        if (electedGlobalState == null) {
            electedGlobalState = nodeState.metaData();
        } else if (nodeState.metaData().version() > electedGlobalState.
version()) {
            electedGlobalState = nodeState.metaData();
        }
    }
}
```

选举索引级元数据:

```
public void performStateRecovery(...)
{
    final Object[] keys = indices.keys;
    //遍历集群的全部索引
    for (int i = 0; i < keys.length; i++) {
        if (keys[i] != null) {
            Index index = (Index) keys[i];
            IndexMetaData electedIndexMetaData = null;
            //遍历请求的全部节点, 对特定索引选择版本号最高的作为该索引元数据
            for (TransportNodesListGatewayMetaState.NodeGatewayMetaState
nodeState : nodesState.getNodes()) {
                IndexMetaData indexMetaData = nodeState.metaData().index
(index);

                if (electedIndexMetaData == null) {
                    electedIndexMetaData = indexMetaData;
                } else if (indexMetaData.getVersion() > electedIndexMetaData.
getVersion()) {
                    electedIndexMetaData = indexMetaData;
                }
            }
        }
    }
}
```



```
    }  
    }  
    }  
}
```

11.4.2 触发 allocation

当上述两个层次的元信息选举完毕，调用 `clusterService.submitStateUpdateTask` 提交一个集群任务，该任务在 `masterService#updateTask` 线程池中执行，实现位于 `GatewayRecoveryListener#onSuccess`。

主要工作是构建集群状态 (`ClusterState`)，其中的内容路由表依赖 `allocation` 模块协助完成，调用 `allocationService.reroute` 进入下一阶段：异步执行分片层元数据的恢复，以及分片分配。`updateTask` 线程结束。

至此，`gateway` 恢复流程结束，集群级和索引级元数据选举完毕，如果存在未分配的主分片，则分片级元数据选举和分片分配正在进行中。

11.5 思考

- 元数据信息是根据版本号选举出来的，而元数据写入成功的条件是“多数”，因此，保证进入 `recovery` 的条件为节点数量为“多数”，可以保证集群级和索引级的一致性。
- 获取各节点存储的元数据，然后根据版本号选举时，仅向具有 `Master` 资格的节点获取元数据。



12 chapter

第 12 章

allocation 模块分析

本章主要分析 allocation 模块的结构和原理，然后以集群启动过程为例分析 allocation 模块的工作过程。

12.1 什么是 allocation

分片分配就是把一个分片指派到集群中某个节点的过程。分配决策由主节点完成，分配决策包含两方面：

- 哪些分片应该分配给哪些节点；
- 哪个分片作为主分片，哪些作为副分片。

对于新建索引和已有索引，分片分配过程也不尽相同。不过不管哪种场景，ES 都通过两个基础组件完成工作：allocators 和 deciders。allocators 尝试寻找最优的节点来分配分片，deciders 则负责判断并决定是否要进行这次分配。

- 对于新建索引，allocators 负责找出拥有分片数最少的节点列表，并按分片数量升序排序，因此分片较少的节点会被优先选择。所以对于新建索引，allocators 的目标就是以更均衡的方式把新索引的分片分配到集群的节点中。然后 deciders 依次遍历 allocators 给出的节点，并判断是否把分片分配到该节点。例如，如果分配过滤规则中禁止节点 A 持有索引 idx 中的任一分片，那么过滤器也阻止把索引 idx 分配到节点 A 中，即便 A 节点是 allocators 从集群负载均衡角度选出的最优节点。需要注意的是，allocators 只关心每个节点上的分片数，而不管每个分片的具体大小。这恰好是 deciders 工作的一部



分，即阻止把分片分配到将超出节点磁盘容量阈值的节点上。

- 对于已有索引，则要区分主分片还是副分片。对于主分片，`allocators` 只允许把主分片指定在已经拥有该分片完整数据的节点上。而对于副分片，`allocators` 则是先判断其他节点上是否已有该分片的数据的副本（即便数据不是最新的）。如果有这样的节点，则 `allocators` 优先把分片分配到其中一个节点。因为副分片一旦分配，就需要从主分片中进行数据同步，所以当节点只拥有分片中的部分数据时，也就意味着那些未拥有的数据必须从主节点中复制得到。这样可以明显地提高副分片的数据恢复速度。

12.2 触发时机

触发分片分配有以下几种情况：

- index 增删；
- node 增删；
- 手工 reroute；
- replica 数量改变；
- 集群重启。

12.3 allocation 模块结构概述

这个复杂的分配过程在 `reroute` 函数中实现：

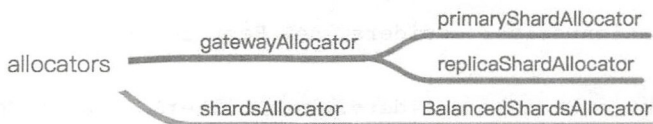
```
AllocationService.reroute
```

此函数对外有两种重载，一种是通过接口调用的手工 `reroute`，另一种是内部模块调用的 `reroute`。本章以内部模块调用的 `reroute` 为例，手工 `reroute` 过程与此类似。

`AllocationService.reroute` 对一个或多个主分片或副分片执行分配，分配以后产生新的集群状态。Master 节点将新的集群状态广播下去，触发后续的流程。对于内部模块调用，返回值为新产生的集群状态，对于手工执行的 `reroute` 命令，返回命令执行结果。

12.4 allocators

目前，`allocators` 的类型如下图所示。



allocators 负责为某个特定的分片分配目的节点。每个 allocator 的主要工作是根据某种逻辑得到一个节点列表，然后调用 deciders 去决策，根据决策结果选择一个目的 node。

allocators 分为 gatewayAllocator 和 shardsAllocator 两种。gatewayAllocator 是为了找到现有分片，shardsAllocator 是根据权重策略在集群的各节点间均衡分片分布。

其中 gatewayAllocator 又分主分片和副分片的 allocator。下面概述每个 allocator 的作用。

- primaryShardAllocator: 找到那些拥有某分片最新数据的节点；
- replicaShardAllocator: 找到磁盘上拥有这个分片数据的节点；
- BalancedShardsAllocator: 找到拥有最少分片个数的节点。

12.5 deciders

目前有下列类型的决策器：

```
public static Collection<AllocationDecider> createAllocationDeciders(...) {
    Map<Class, AllocationDecider> deciders = new LinkedHashMap<>();
    addAllocationDecider(deciders, new MaxRetryAllocationDecider(settings));
    addAllocationDecider(deciders, new ResizeAllocationDecider(settings));
    addAllocationDecider(deciders, new ReplicaAfterPrimaryActiveAllocation-
Decider(settings));
    addAllocationDecider(deciders, new RebalanceOnlyWhenActiveAllocation-
Decider(settings));
    addAllocationDecider(deciders, new ClusterRebalanceAllocationDecider
(settings, clusterSettings));
    addAllocationDecider(deciders, new ConcurrentRebalanceAllocationDecider
(settings, clusterSettings));
    addAllocationDecider(deciders, new EnableAllocationDecider(settings,
clusterSettings));
    addAllocationDecider(deciders, new NodeVersionAllocationDecider(settings));
```





```
        addAllocationDecider(deciders, new SnapshotInProgressAllocationDecider
(settings));
        addAllocationDecider(deciders, new RestoreInProgressAllocationDecider
(settings));
        addAllocationDecider(deciders, new FilterAllocationDecider(settings,
clusterSettings));
        addAllocationDecider(deciders, new SameShardAllocationDecider(settings,
clusterSettings));
        addAllocationDecider(deciders, new DiskThresholdDecider(settings,
clusterSettings));
        addAllocationDecider(deciders, new ThrottlingAllocationDecider(settings,
clusterSettings));
        addAllocationDecider(deciders, new ShardsLimitAllocationDecider(settings,
clusterSettings));
        addAllocationDecider(deciders, new AwarenessAllocationDecider(settings,
clusterSettings));
        return deciders.values();
    }
}
```

它们继承自 `AllocationDecider`，需要实现的接口有：

- * `canRebalance`，给定分片是否可以“re-balanced”到给定 allocation;
- * `canAllocate`，给定分片是否可以分配到给定节点;
- * `canRemain`，给定分片是否可以保留在给定节点;
- * `canForceAllocatePrimary`，给定主分片是否可以强制分配在给定节点。

这些 `deciders` 在 `ClusterModule#createAllocationDeciders` 中全部添加进去，`decider` 运行之后可能产生的结果有以下几种：

ALWAYS、YES、NO、THROTTL

这些 `deciders` 大致可以分为以下几类。

12.5.1 负载均衡类

- `SameShardAllocationDecider`
避免主副分片分配到同一个节点。





- AwarenessAllocationDecider

感知分配器，感知服务器、机架等，尽量分散存储 shard。

有两种参数用于调整：

```
cluster.routing.allocation.awareness.attributes: rack_id
cluster.routing.allocation.awareness.attributes: zone
```

- ShardsLimitAllocationDecider

同一个节点上允许存在的同一个 index 的 shard 数目。

12.5.2 并发控制类

- ThrottlingAllocationDecider

recovery 阶段的限速配置，包括：

```
cluster.routing.allocation.node_concurrent_recoveries
cluster.routing.allocation.node_initial_primaries_recoveries
cluster.routing.allocation.node_concurrent_incoming_recoveries
cluster.routing.allocation.node_concurrent_outgoing_recoveries
```

- ConcurrentRebalanceAllocationDecider

rebalance 并发控制，可以通过下面的参数配置：

```
cluster.routing.allocation.cluster_concurrent_rebalance
```

- DiskThresholdDecider

根据磁盘空间进行决策的分配器。

12.5.3 条件限制类

- RebalanceOnlyWhenActiveAllocationDecider

所有 shard 都处在 active 状态下，才可以执行 rebalance 操作。

- FilterAllocationDecider

可以调整的参数如下，可以通过接口动态设置：





```
index.routing.allocation.require.*【必须】
index.routing.allocation.include.*【允许】
index.routing.allocation.exclude.*【排除】
cluster.routing.allocation.require.*
cluster.routing.allocation.include.*
cluster.routing.allocation.exclude.*
```

配置的目标为节点 IP 或节点名等。cluster 级别设置会覆盖 index 级别设置。

- ReplicaAfterPrimaryActiveAllocationDecider

保证只在主分片分配完毕才开始分配分片副本。

- ClusterRebalanceAllocationDecider

通过集群中 active 的 shard 状态来决定是否可以执行 rebalance, 通过下面的配置控制, 可以动态生效:

```
cluster.routing.allocation.allow_rebalance
```

可配置的值如下表所示。

取 值	含 义
indices_all_active	当集群所有的节点分配完毕, 才认定集群 rebalance 完成 (默认)
indices primaries_active	只要所有主分片分配完毕, 就可以认定集群 rebalance 完成
always	即使当主分片和分片副本都没有分配, 也允许 rebalance 操作

12.6 核心 reroute 实现

reroute 中主要实现两种 allocator:

- gatewayAllocator, 用于分配现实已存在的分片, 从磁盘中找到它们;
- shardsAllocator, 用于平衡分片在节点间的分布。

```
private void reroute(RoutingAllocation allocation) {
    if (allocation.routingNodes().unassigned().size() > 0) {
        removeDelayMarkers(allocation);
        //gateway 分配器
        gatewayAllocator.allocateUnassigned(allocation);
    }
    //分片均衡分配器
```





```
shardsAllocator.allocate(allocation);  
}
```

reroute 流程全部运行于 masterService#updateTask 线程。

12.6.1 集群启动时 reroute 的触发时机

gateway 结束前调用 submitStateUpdateTask 提交任务，任务被 clusterService 放入队列，在 Master 节点顺序执行。

执行到任务中的：

```
allocationService.reroute
```

收集各个节点的 shard 元数据，待某个 shard 的 Response 从所有节点全部返回后，执行 finishHim()，然后对收集到的数据进行处理：

```
AsyncShardFetch#processAsyncFetch
```

allocationService.reroute 执行完毕返回新的集群状态。

下面以集群启动时 gateway 之后的 reroute 为背景分析流程。

12.6.2 流程分析

gateway 阶段恢复的集群状态中，我们已经知道集群一共有多少个索引，每个索引的主副分片各有多少个，但是不知道它们位于哪个节点，现在需要找到它们都位于哪个节点。集群完全重启的初始状态，所有分片都被标记为未分配状态，此处也被称作分片分配过程。因此分片分配的概念不仅仅是分配一个全新分片。

对于索引某个特定分片的分配过程中，先分配其主分片，后分配其副分片。

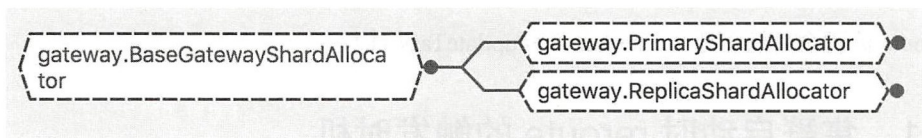
12.6.3 gatewayAllocator

gatewayAllocator 分为主分片和副分片分配器：

```
primaryShardAllocator.allocateUnassigned(allocation);  
replicaShardAllocator.processExistingRecoveries(allocation);  
replicaShardAllocator.allocateUnassigned(allocation);
```



它们都继承自 BaseGatewayShardAllocator，如下图所示。



主分片分配器

primaryShardAllocator.allocateUnassigned 函数实现整个分配过程。

主分片分配器与副分片分配器都继承自 BaseGatewayShardAllocator，执行相同的 allocateUnassigned 函数，只是执行 makeAllocationDecision 时，主副分片分配器各自执行自己的策略。

allocateUnassigned 的流程是：遍历所有 unassigned shard，依次处理，通过 decider 决策分配，期间可能需要 fetchData 获取这个 shard 对应的元数据。如果决策结果为 YES，则将其初始化。

```

public void allocateUnassigned(RoutingAllocation allocation) {
    //遍历未分配的分片
    while (unassignedIterator.hasNext()) {
        final ShardRouting shard = unassignedIterator.next();
        //调用主分片分配器的 makeAllocationDecision 进行决策
        final AllocateUnassignedDecision allocateUnassignedDecision =
            makeAllocationDecision(shard, allocation, logger);

        //根据决策器结果决定是否初始化分片
        if (allocateUnassignedDecision.getAllocationDecision() ==
            AllocationDecision.YES) {
            unassignedIterator.initialize(...);
        } else {
            unassignedIterator.removeAndIgnore(...);
        }
    }
}

```

主副分片执行相同的 unassignedIterator.initialize 函数，将分片的 unassigned 状态改为 initialize 状态：



```

public ShardRouting initializeShard(...) {
    //初始化一个 ShardRouting
    ShardRouting initializedShard = unassignedShard.initialize(nodeId,
existingAllocationId, expectedSize);
    //添加到目的节点的 shard 列表
    node(nodeId).add(initializedShard);
    addRecovery(initializedShard);
    //添加到 assigned shards 列表
    assignedShardsAdd(initializedShard);
    //设置状态已更新
    routingChangesObserver.shardInitialized(unassignedShard,
initializedShard);
    return initializedShard;
}

```

在 `routingChangesObserver.shardInitialized` 中设置 `RoutingNodes` 已更新。更新的内容大约就是某个 shard 被分配到了某个节点, 这个 shard 是主还是副, 副的话会设置 `recoverySource` 为 `PEER`, 但只是一个类型, 并没有告诉节点 `recovery` 的时候从哪个节点恢复, 节点恢复时自己从集群状态中的路由表中查找。

`reroute` 完成后构建新的集群状态:

```

protected ClusterState reroute(final ClusterState clusterState, String
reason, boolean debug) {
    reroute(allocation);
    if (allocation.routingNodesChanged() == false) {
        return clusterState;
    }
    return buildResultAndLogHealthChange(clusterState, allocation, reason);
}

```

然后, `Master` 把新的集群状态广播下去, 当数据节点发现某个分片分配给自己, 开始执行分片的 `recovery`。

PrimaryShardAllocator#makeAllocationDecision

主分片分配器的 `makeAllocationDecision` 过程返回指定的分片是否可以被分配, 如果还没有这个分片的信息, 则向集群的其他节点去请求该信息; 如果已经有了, 则根据 `decider` 进行决策。



首次进入函数时，还没有任何分片的元信息，发起向集群所有数据节点获取某个 shard 元信息的 `fetchData` 请求。

之所以把请求发到所有节点，是因为它不知道哪个节点有这个 shard 的数据。集群启动的时候，遍历所有 shard，再对每个 shard 向所有数据节点发 `fetchData` 请求。如果集群有 100 个节点、1000 个分片，则总计需要请求 $100 \times 1000 = 100000$ 次。虽然是异步的，但仍然存在效率问题。当 ES 集群规模比较大、分片数非常多的时候，这个请求的总量就会很大。

1. 向各节点发起 `fetchData` 请求

从所有数据节点异步获取某个特定分片的信息，没有超时设置。

```
void asyncFetch(final DiscoveryNode[] nodes, long fetchingRound) {
    //nodes 为节点列表
    action.list(shardId, nodes, new ActionListener<BaseNodesResponse<T>>() {
        //收到成功的回复
        public void onResponse(BaseNodesResponse<T> response) {
            processAsyncFetch(response.getNodes(), response.failures(),
fetchingRound);
        }
        //收到失败的回复
        public void onFailure(Exception e) {
            List<FailedNodeException> failures = new ArrayList<>
(nodes.length);
            for (final DiscoveryNode node: nodes) {
                failures.add(...);
            }
            processAsyncFetch(null, failures, fetchingRound);
        }
    });
}
```

请求节点 shard 元信息的 action 为：

```
internal:gateway/local/started_shards
```

遍历发送的过程主要实现位于：

```
TransportNodesAction.AsyncAction#start
```



2. 数据节点的响应

对端对此响应的模块为：

```
gateway.TransportNodesListGatewayStartedShards#nodeOperation
```

其中读取本地 shard 元数据返回请求方。

3. 收集返回结果并处理

对于一个特定 shard，当 Response 达到期望数量（发出请求时的节点数）时执行 finishHim，调用处理模块：

```
AsyncShardFetch#processAsyncFetch
```

在这里实现收到各节点返回的 shard 级别元数据的对应处理，将 Response 信息放到 this.cache 中，下次“reroute”的时候从 cache 里取，然后再次执行 reroute。

```
reroute(shardId, "post_response");
//接着调用-->
routingService.reroute("async_shard_fetch");
```

主要实现是对当前 ClusterState 提交一个任务，再次执行 allocationService.reroute，此时 this.cache 中已经有了 shard 元数据，进入主分片分配过程，依据这些元信息选择主分片。

4. 主分片选举实现

ES 5 之后的主分片选举与之前的版本机制是不一样的。ES 5 之前的版本依据分片元数据的版本号对比实现，选择分片元信息中版本号高的分片来选举主分片，ES 5 及之后的版本依据 allocation id 从 inSyncAllocationIds 列表选择一个作为主分片。这种改变的主要原因是依据版本号无法保证版本号最高的分片一定被选为主分片。例如，当前只有一个活跃分片，那它一定被选为主分片，而拥有最新数据的分片尚未启动。具体请参考“数据模型”章节。

在 makeAllocationDecision 函数中，从 inSyncAllocationIds 集合中找到活跃的 shard，得到一个列表，把这个列表中的节点依次执行 decider，决策结果：

```
//根据 shard id 从集群状态获取同步 ID 列表
final Set<String> inSyncAllocationIds = indexMetaData.inSyncAllocationIds
(unassignedShard.id());
//构建匹配同步 ID 的目标节点列表
final NodeShardsResult nodeShardsResult = buildNodeShardsResult(unassignedShard,
```



```

        snapshotRestore, allocation.getIgnoreNodes(unassignedShard.shardId()),
        inSyncAllocationIds, shardState, logger);
    //对上一步得到的列表中的节点依次执行 decider, 决策结果
    NodesToAllocate nodesToAllocate = buildNodesToAllocate(
        allocation, nodeShardsResult.orderedAllocationCandidates,
        unassignedShard, false
    );

```

具体的决策过程会依次遍历全部 decider:

```

    public Decision canAllocate(ShardRouting shardRouting, RoutingNode node,
        RoutingAllocation allocation) {
        //遍历全部 decider
        for (AllocationDecider allocationDecider : allocations) {
            Decision decision = allocationDecider.canAllocate(shardRouting,
                node, allocation);
            if (decision == Decision.NO) {
                if (!allocation.debugDecision()) {
                    return decision;
                } else {
                    ret.add(decision);
                }
            } else if (decision != Decision.ALWAYS
                && (allocation.getDebugMode() != EXCLUDE_YES_DECISIONS ||
                    decision.type() != Decision.Type.YES)) {
                ret.add(decision);
            }
        }
        return ret;
    }

```

只要有一个 decider 拒绝, 就拒绝执行本次分配。决策之后的结果可能会有多个节点, 取第一个。至此, 主分片选取完成。

cluster.routing.allocation.enable 对主分片分配的影响

在集群完全重启的操作流程中, 要求把这个选项先设置为 none, 然后重启集群。分片分配时, 在 EnableAllocationDecider 中对这个选项进行判断和实施, 主分片的分配会被拦截吗? 答案是肯定的, 主分片被这个 decider 拦截。但是在主分片的分配过程中有另外一层逻辑: 如果被

decider 拦截, 返回 NO, 则尝试强制分配。给 buildNodesToAllocate 的最后一个参数传入 true, 接下来尝试强制分配, 逻辑如下:

```
public Decision canForceAllocatePrimary(ShardRouting shardRouting,
RoutingNode node, RoutingAllocation allocation) {
    Decision decision = canAllocate(shardRouting, node, allocation);
    if (decision.type() == Type.NO) {
        // On a NO decision, by default, we allow force allocating the primary.
        return allocation.decision(Decision.YES,
            decision.label(),
            "primary shard [%s] allowed to force allocate on node [%s]",
            shardRouting.shardId(), node.nodeId());
    } else {
        // On a THROTTLE/YES decision, we use the same decision instead of
        forcing allocation
        return decision;
    }
}
```

如果 decider 返回 NO, 则直接设置成 YES, 这种情况只在分配一个磁盘上已经存在的 unassigned primary shards 时出现。

副分片分配器

与主分片分配器同理, replicaShardAllocator.allocateUnassigned 函数实现了副分片的分配过程。其 allocateUnassigned 过程与 primaryShardAllocator.allocateUnassigned 执行的是基类的同一个函数。

ReplicaShardAllocator#makeAllocationDecision

副分片决策过程中也需要“fetchData”, 只不过主分片分配节点已经“fetch”过, 可以直接从结果中获取。但是在“fetchData”之前先运行一遍 allocation.deciders().canAllocate, 来判断是否至少可以在一个 node 上分配 (canBeAllocatedToAtLeastOneNode), 如果分配不了就省略后面的逻辑了, 例如, 其主分片尚未就绪等。

然后根据“fetchData”到的 shard 元信息, 分配到已经拥有这个 shard 副本的节点, 如果没有相关节点, 则判断是否需要“delay”, 否则返回 NOT_TAKEN。分配成功之后一样进入 INIT 过程。

是否“delay”由下面的配置项控制, 可以动态调整:

```
index.unassigned.node_left.delayed_timeout
```

12.6.4 shardsAllocator

shardsAllocator 由 BalancedShardsAllocator 类完成，其中有三个配置项用于控制权重：

```
cluster.routing.allocation.balance.index  
cluster.routing.allocation.balance.shard  
cluster.routing.allocation.balance.threshold
```

这些权重参数的具体作用建议参考 *Mastering Elasticsearch* 或官方手册。

具体实现如下：

```
public void allocate(RoutingAllocation allocation) {  
    final Balancer balancer = new Balancer(logger, allocation, weightFunction,  
threshold);  
    balancer.allocateUnassigned();  
    balancer.moveShards();  
    balancer.balance();  
}
```

allocateUnassigned

根据权重算法和 decider 决定把 shard 分配到哪个节点。同样将决策后的分配信息更新到集群状态，由 Master 广播下去。

moveShards

对状态为 started 的分片根据 decider 来判断是否需要“move”，move 过程中此 shard 的状态被设置为 RELOCATING，在目标上创建这个 shard 时状态为 INITIALIZING，同时版本号会加 1。

balance

根据权重函数平衡集群模型上的节点。

12.7 从 gateway 到 allocation 流程的转换

两者之间没有明显的界限，gateway 的最后一步执行 reroute，等待这个函数返回，然后打印 gateway 选举结果的日志，集群完全重启时，reroute 向各节点发起的询问 shard 级元数据的操

作基本还没执行完, 因此一般只有少数主分片被选举完了, gateway 流程的结束只是集群级和索引级的元数据已选举完毕, 主分片的选举正在进行中。

12.8 从 allocation 流程到 recovery 流程的转换

makeAllocationDecision 成功后, unassignedIterator.initialize 初始化这个 shard, 创建一个新的 ShardRouting 对象, 把相关信息添加到集群状态, 设置 routingChangesObserver 为已经发生变化, 后面的流程就会把新的集群状态广播出去。到此, reroute 函数执行完毕。

节点收到广播下来的集群状态, 进入 IndicesClusterStateService#applyClusterState 处理所有相关操作, 其中, createOrUpdateShards 执行到 createShard 时, 准备 recovery 相关信息:

```
private void createShard(DiscoveryNodes nodes, RoutingTable routingTable,
    ShardRouting shardRouting, ClusterState state) {
    if (shardRouting.recoverySource().getType() == Type.PEER) {
        sourceNode = findSourceNodeForPeerRecovery(logger, routingTable,
            nodes, shardRouting);
    }

    try {
        RecoveryState recoveryState = new RecoveryState(shardRouting,
            nodes.getLocalNode(), sourceNode);
        indicesService.createShard(...);
    } catch (Exception e) {
        failAndRemoveShard(shardRouting, true, "failed to create shard", e,
            state);
    }
}
```

接下来 IndicesService.createShard 开始执行 recovery:

```
public IndexShard createShard(...) throws IOException {
    IndexService indexService = indexService(shardRouting.index());
    IndexShard indexShard = indexService.createShard(shardRouting,
        globalCheckpointSyncer);
    indexShard.addShardFailureCallback(onShardFailure);
    indexShard.startRecovery(recoveryState, recoveryTargetService,
        recoveryListener, repositoriesService,
```

```
(type, mapping) -> {
    try {
        client.admin().indices().preparePutMapping()
            .setConcreteIndex(shardRouting.index()) // concrete
index - no name clash, it uses uuid
            .setType(type)
            .setSource(mapping.source().string(), XContentType.JSON)
            .get();
    } catch (IOException ex) {
        throw new ElasticsearchException("failed to stringify
mapping source", ex);
    }
    }, this);
    return indexShard;
}
```

12.9 思考

- 请求分片信息的 `fetchData` 请求效率低，可以借鉴 HDFS 的上报块状态流程。
- 不需要等所有主分片都分配完才执行副分片的分配。每个分片有自己的分配流程。
- 不需要等所有分片都分配完才执行 `recovery` 流程。
- 主分片不需要等副分片分配成功才进入主分片的 `recovery`，主副分片有自己的 `recovery` 流程。

13 chapter

第 13 章

Snapshot 模块分析

快照模块是 ES 备份、迁移数据的重要手段。它支持增量备份，支持多种类型的仓库存储。本章我们先来看看如何使用快照，以及它的一些细节特性，然后分析创建、删除及取消快照的实现原理。

仓库用于存储快照，支持共享文件系统(例如，NFS)，以及通过插件支持的 HDFS、AmazonS3、Microsoft Azure、Google GCS。

在跨版本支持方面，可以支持不跨大版本的快照和恢复。

- 在 6.x 版本中创建的快照可以恢复到 6.x 版本；
- 在 2.x 版本中创建的快照可以恢复到 5.x 版本；
- 在 1.x 版本中创建的快照可以恢复到 2.x 版本。

相反，1.x 版本创建的快照不可以恢复到 5.x 版本和 6.0 版本，2.x 版本创建的快照不可以恢复到 6.x 版本。

升级集群前建议先通过快照备份数据。跨越大版本的数据迁移可以考虑使用 `reindex` API。

当需要迁移数据时，可以将快照恢复到另一个集群。快照不仅可以对索引备份，还可以将模板一起保存。恢复到的目标集群不需要相同的节点规模，只要它的存储空间足够容纳这些数据即可。

要使用快照，首先应该注册仓库。快照存储于仓库中。

13.1 仓库

仓库用于存储创建的快照。建议为每个大版本创建单独的快照存储库。如果使用多个集群注册相同的快照存储库，那么最好只有一个集群对存储库进行写操作。连接到该存储库的其他集群都应该将该存储库设置为 `readonly` 模式。

使用下面的命令注册一个仓库：

```
curl -X PUT "localhost:9200/_snapshot/my_backup" -H 'Content-Type: application/json' -d'
{
  "type": "fs",
  "settings": {
    "location": "/mnt/my_backup"
  }
}
```

本例中，注册的仓库名称为 `my_backup`，`type` 为 `fs`，指定仓库类型为共享文件系统。共享文件系统支持的配置如下表所示。

参 数	简 介
<code>location</code>	指定了一个已挂载的目的地址
<code>compress</code>	是否开启压缩。压缩仅对元数据进行（ <code>mapping</code> 及 <code>settings</code> ），不对数据文件进行压缩，默认为 <code>true</code>
<code>chunk_size</code>	传输文件时数据被分解为块，此处配置块大小，单位为字节，默认为 <code>null</code> （无限块大小）
<code>max_snapshot_bytes_per_sec</code>	快照操作时节点间限速值，默认为 40MB
<code>max_restore_bytes_per_sec</code>	从快照恢复时节点间限速值，默认为 40MB
<code>readonly</code>	设置仓库属性为只读，默认为 <code>false</code>

要获取某个仓库配置信息，可以使用下面的 API：

```
curl -X GET "localhost:9200/_snapshot/my_backup"
```

返回信息如下：

```
{
  "my_backup": {
```

```

    "type": "fs",
    "settings": {
      "location": "/mnt/my_backup"
    }
  }
}

```

要获取多个存储库的信息，可以指定一个以逗号分隔的存储库列表，还可以在指定存储库名称时使用 “*” 通配符。例如：

```
curl -X GET "localhost:9200/_snapshot/repo*,*backup*"
```

要获取当前全部仓库的信息，可以省略仓库名称，使用 `_all`：

```
curl -X GET "localhost:9200/_snapshot"
```

或

```
curl -X GET "localhost:9200/_snapshot/_all"
```

可以使用下面的命令从仓库中删除快照：

```
curl -X DELETE "localhost:9200/_snapshot/my_backup/snapshot_1"
```

可以使用下面的命令删除整个仓库：

```
curl -X DELETE "localhost:9200/_snapshot/my_backup"
```

当仓库被删除时，ES 只是删除快照的仓库位置引用信息，快照本身没有删除。

共享文件系统

当使用共享文件系统时，需要将同一个共享存储挂载到集群每个节点的同一个挂载点（路径），包括所有数据节点和主节点。然后将这个挂载点配置到 `elasticsearch.yml` 的 `path.repo` 字段。例如，挂载点为 `/mnt/my_backup`，那么在 `elasticsearch.yml` 中应该添加如下配置：

```
path.repo: ["/mnt/my_backups"]
```

`path.repo` 配置以数组的形式支持多个值。如果配置多个值，则不像 `path.data` 一样同时使用这些路径，相反，应该为每个挂载点注册不同的仓库。例如，一个挂载点存储空间不足以容纳

集群所有数据，可使用多个挂载点，同时注册多个仓库，将数据分开快照到不同的仓库。

path.repo 支持微软的 UNC 路径，配置格式如下：

```
path.repo: ["\\MY_SERVER\\Snapshots"]
```

当配置完毕，需要重启所有节点使之生效。然后就可以通过仓库 API 注册仓库，执行快照了。

使用共享存储的优点是跨版本兼容性好，适合迁移数据。缺点是存储空间较小。如果使用 HDFS，则受限与插件使用的 HDFS 版本。插件版本要匹配 ES，而这个匹配的插件使用固定版本的 HDFS 客户端。一个 HDFS 客户端只支持写入某些兼容版本的 HDFS 集群。

13.2 快照

13.2.1 创建快照

存储库可以包含同一集群的多个快照。每个快照有唯一的名称标识。通过以下命令在 my_backup 仓库中为全部索引创建名为 snapshot_1 的快照：

```
curl -X PUT "localhost:9200/_snapshot/my_backup/snapshot_1?wait_for_completion=true"
```

wait_for_completion 参数是可选项，默认情况下，快照命令会立即返回，任务在后台执行。如果想等待任务完成 API 才返回，则可以将 wait_for_completion 参数设置为 true，默认为 false。

上述命令会为所有 open 状态的索引创建快照。如果想对部分索引执行快照，则可以在请求的 indices 参数中指定：

```
curl -X PUT "localhost:9200/_snapshot/my_backup/snapshot_2?wait_for_completion=true" -H 'Content-Type: application/json' -d'
{
  "indices": "index_1,index_2",
  "ignore_unavailable": true,
  "include_global_state": true
}
```

indices 字段支持多索引语法，index_*完整的语法参考：<https://www.elastic.co/guide/en/>

elasticsearch/reference/current/multi-index.html。

另外两个参数：

- `ignore_unavailable`，跳过不存在的索引。默认为 `false`，因此默认情况下遇到不存在的索引快照失败。
- `include_global_state`，不快照集群状态。默认为 `false`。注意，集群设置和模板保存在集群状态中，因此默认情况下不快照集群设置和模板，但是一般情况下我们需要将这些信息一起保存。

快照操作在主分片上执行。快照执行期间，不影响集群正常的读写操作。在快照开始前，会执行一次 `flush`，将操作系统内存“`cache`”的数据刷盘。因此通过快照可以获取从成功执行快照的时间点开始，磁盘中存储的 Lucene 数据，不包括后续的新增内容。但是每次快照过程是增量的，下一次快照只会包含新增内容。

可以在任何时候为集群创建一个快照过程，无论集群健康是 `Green`、`Yellow`，还是 `Red`。执行快照期间，被快照的分片不能移动到另一个节点，这可能会干扰重新平衡过程和分配过滤（`allocation filtering`）。这种分片迁移只可以在快照完成时进行。

快照开始后，可以用快照信息 API 和 `status API` 来监控进度。

13.2.2 获取快照信息

当快照开始后，使用下面的 API 来获取快照的信息：

```
curl -X GET "localhost:9200/_snapshot/my_backup/snapshot_1"
```

返回信息摘要如下：

```
{
  "snapshots": [
    {
      "snapshot": "snapshot_1",
      "version": "6.1.2",
      "indices": [
        "website"
      ],
      "state": "SUCCESS",
      "start_time": "2018-05-15T03:40:06.571Z",
      "end_time": "2018-05-15T07:53:40.977Z",
    }
  ]
}
```



```
    "duration_in_millis": 15214406,
    "failures": [],
    "shards": {
      "total": 6,
      "failed": 0,
      "successful": 6
    }
  }
}
```

主要是开始结束时间、集群版本、当前阶段、成功及失败情况等基本信息。快照执行期间会经历以下几个阶段，如下表所示。

阶 段	简 介
IN_PROGRESS	快照正在运行
SUCCESS	快照创建完成，并且所有分片都存储成功
FAILED	快照创建失败，没有存储任何数据
PARTIAL	全局集群状态已储存，但至少有一个分片的数据没有存储成功。在返回的 <code>failure</code> 字段中包含了关于未正确处理分片的详细信息
INCOMPATIBLE	快照与当前集群版本不兼容

使用下面的命令可以获取多个快照信息：

```
curl -X GET "localhost:9200/_snapshot/my_backup/snapshot_*,other_snapshot"
```

以及获取指定仓库下的全部快照信息：

```
curl -X GET "localhost:9200/_snapshot/my_backup/_all"
```

如果一些快照不可用导致命令失败，则可以通过设置布尔参数 `ignore_unavailable` 来返回当前可用的所有快照。

可以使用下面的命令来查询正在运行中的快照：

```
curl -X GET "localhost:9200/_snapshot/my_backup/_current"
```

13.2.3 快照 status

_status API 用于返回快照的详细信息。

可以使用下面的命令来查询当前正在运行的全部快照的详细状态信息：

```
curl -X GET "localhost:9200/_snapshot/_status"
```

返回的信息摘要如下：

```
"stats": {  
  "number_of_files": 31,  
  "processed_files": 31,  
  "total_size_in_bytes": 33802,  
  "processed_size_in_bytes": 33802,  
  "start_time_in_millis": 1526355676967,  
  "time_in_millis": 15144003  
}
```

主要是已处理的文件数和字节数等进度信息，但没有计算成百分比的形式。

使用下面的命令可以返回特定仓库中正在运行的所有快照的信息：

```
curl -X GET "localhost:9200/_snapshot/my_backup/_status"
```

如果同时指定仓库名称和快照 ID，则此命令将返回指定快照的状态信息，即使它已经执行完成：

```
curl -X GET "localhost:9200/_snapshot/my_backup/snapshot_1,snapshot_2/_status"
```

13.2.4 取消、删除快照和恢复操作

在设计上，快照和恢复在同一个时间点只允许运行一个快照或一个恢复操作。如果想终止正在进行的快照操作，则可以使用删除快照命令来终止它。删除快照操作将检查当前快照是否正在运行，如果正在运行，则删除操作会先停止快照，然后从仓库中删除数据。如果是已完成的快照，则直接从仓库中删除快照数据。

```
curl -X DELETE "localhost:9200/_snapshot/my_backup/snapshot_1"
```

恢复操作使用标准分片恢复机制。因此，如果要取消正在运行的恢复，则可以通过删除正在恢复的索引来实现。注意，索引数据将全部删除。

13.3 从快照恢复

要恢复一个快照，目标索引必须处于关闭状态。

使用下面的命令恢复一个快照：

```
curl -X POST "localhost:9200/_snapshot/my_backup/snapshot_1/_restore"
```

默认情况下，快照中的所有索引都被恢复，但不恢复集群状态。通过调节参数，可以有选择地恢复部分索引和全局集群状态。索引列表支持多索引语法。例如：

```
curl -X POST "localhost:9200/_snapshot/my_backup/snapshot_1/_restore" -H
'Content-Type: application/json' -d'
{
  "indices": "index_1,index_2",
  "ignore_unavailable": true,
  "include_global_state": true,
  "rename_pattern": "index_(.+)",
  "rename_replacement": "restored_index_$1"
}
```

具体参数介绍如下表所示。

参 数	简 介
indices	要恢复的索引列表。支持多索引语法（multi index syntax）
ignore_unavailable	与快照时含义相同
include_global_state	是否恢复集群状态，默认为 false。设置为 true 时，快照中的模板被恢复，如果当前集群存在同名的模板，则会被覆盖。集群设置（persistent settings）同样被恢复
rename_pattern	如下
rename_replacement	与上一个参数配合，通过正则表达式对恢复的索引重命名
partial	是否允许在遇到错误时恢复部分数据，默认为 false

恢复完成后，当前集群与快照同名的索引、模板会被覆盖。在集群中存在，但快照中不存

在的索引、索引别名、模板不会被删除。因此恢复并非同步成与快照一致。

13.3.1 部分恢复

默认情况下，在恢复操作时，如果参与恢复的一个或多个索引在快照中没有可用分片，则整个恢复操作失败。这可能是因为创建快照时一些分片备份失败导致的。可以通过设置 `partial` 参数为 `true` 来尽可能恢复。但是，只有备份成功的分片才会成功恢复，丢失的分片将被创建一个空的分片。

13.3.2 恢复过程中更改索引设置

大多数的索引设置可以在恢复过程中被重写。例如，下面的指令将在恢复索引 `index_1` 时不创建任何副本，以及采用默认的索引刷新间隔：

```
curl -X POST "localhost:9200/_snapshot/my_backup/snapshot_1/_restore" -H
'Content-Type: application/json' -d'
{
  "indices": "index_1",
  "index_settings": {
    "index.number_of_replicas": 0
  },
  "ignore_index_settings": [
    "index.refresh_interval"
  ]
}
```

有一些设置不能在恢复时修改，比如 `index.number_of_shards`。

13.3.3 监控恢复进度

恢复过程是基于 ES 标准恢复机制的，因此标准的恢复监控服务可以用来监视恢复的状态。当执行集群恢复操作时通常会进入 `Red` 状态，这是因为恢复操作是从索引的主分片开始的，在此期间主分片状态变为不可用，因此集群状态表现为 `Red`。一旦 ES 主分片恢复完成，整个集群的状态将被转换成 `Yellow`，并且开始创建所需数量的副分片。一旦创建了所有必需的副分片，集群转换到 `Green` 状态。

查看集群的健康情况只是在恢复过程中比较高级别的状态，还可以通过使用 `indices recovery` 与 `cat recovery` 的 API 来获得更详细的恢复过程信息与索引的当前状态信息。

13.4 创建快照的实现原理

在快照的实现原理中我们重点关注几个问题：快照是如何实现的？增量过程是如何实现的？为什么删除旧快照不影响其他快照？

ES 的快照创建是基于 Lucene 快照实现的。但是 Lucene 中的快照概念与 ES 的并不相同。

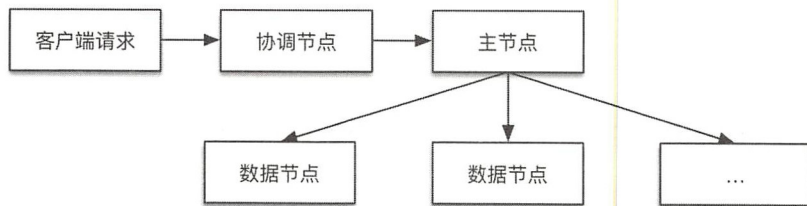
Lucene 快照是对最后一个提交点的快照，一次快照包含最后一次提交点的信息，以及全部分段文件。因此这个快照实际上就是对已刷盘数据的完整的快照。注意 Lucene 中没有增量快照的概念。每一次都是对整个 Lucene 索引完整快照，它代表这个 Lucene 索引的最新状态。之所以称为快照，是因为从创建一个 Lucene 快照开始，与此快照相关的物理文件都保证不会删除。在 Lucene 中，快照通过 `SnapshotDeletionPolicy` 实现。从 Lucene 2.3 版本开始支持。

你可能还记得，在副分片的恢复过程中，也需要对主分片创建 Lucene 快照，然后复制数据文件。

因此总结来说：

- Lucene 快照负责获取最新的、已刷盘的分段文件列表，并保证这些文件不被删除，这个文件列表就是 ES 要执行复制的文件。
- ES 负责数据复制、仓库管理、增量备份，以及快照删除。

创建快照的整体过程如下图所示。



ES 创建快照的过程涉及 3 种类型的节点：

- **协调节点**，接收客户端请求，转发到主节点。
- **主节点**，将创建快照相关的请求信息放到集群状态中广播下去，数据节点收到后执行数据复制。同时负责在仓库中写入集群状态数据。
- **数据节点**，负责将 Lucene 文件复制到仓库，并在数据复制完毕后清理仓库中与任何快

照都不相关的文件。由于数据分布在各个数据节点，因此复制操作必须由数据节点执行。每个数据节点将快照请求中本地存储的主分片复制到仓库。

快照过程是对 Lucene 物理文件的复制过程，一个 Lucene 索引由许多不同类型的文件组成，完整的介绍可以参考 Lucene 官方手册，当前版本的地址为：http://lucene.apache.org/core/7_3_1/index.html。

如果数据节点在执行快照过程中异常终止，例如，I/O 错误，进程被“kill”，服务器断电等异常，则这个节点上执行的快照尚未成功，当这个节点重新启动，不会继续之前的数据复制流程。对于整个快照进程来说，最终结果是部分成功、部分失败。快照信息中会记录失败的节点和分片，以及与错误相关的原因。

13.4.1 Lucene 文件格式简介

1. 定义

Lucene 中基本的概念包括 index、document、field 和 term。一个 index 包含一系列的 document：

- 一个 document 是一系列的 fields；
- 一个 field 是一系列命名的 terms；
- 一个 term 是一系列 bytes。

2. 分段

Lucene 索引可能由多个分段（segment）组成，每个分段是一个完全独立的索引，可以独立执行搜索。有两种情况产生新的分段：

- refresh 操作产生一个 Lucene 分段。为新添加的 documents 创建新的分段。
- 已存在的分段合并，产生新分段。

一次对 Lucene 索引的搜索需要搜索全部分段。

3. 文件命名规则

属于一个段的所有文件都具有相同的名称和不同的扩展名。当使用复合索引文件（默认）时，除 .si write.lock .del 外的其他文件被合并压缩成单个 .cfs 文件。

文件名不会被重用，也就是说，任何文件被保存到目录中时，它有唯一的文件名。这是通过简单的生成方法实现的，例如，第一个分段文件名为 segments_1，接下来是 segments_2。

4. 文件扩展名摘要

下表总结了 Lucene 中文件的名称和扩展名。

名 称	扩 展 名	简 介
Segments File	segments_N	保存提交点信息。通过这个文件可以知道当前的 Lucene 索引都有哪些分段
Lock File	write.lock	写锁，防止多个 IndexWriters 写同一个文件
Segment Info	.si	存储分段元数据信息
Compound File	.cfs、.cfe	复合索引文件，使用复合文件的好处是减少许多文件句柄（文件描述符）
Fields	.fnm	保存 fields 的相关信息
Field Index	.fdx	保存指向 field data 的指针
Field Data	.fdt	文档存储的 fields 的值
Term Dictionary	.tim	term 词典，存储 term 信息
Term Index	.tip	Term Dictionary 的索引
Frequencies	.doc	包含词频的 doc 列表
Positions	.pos	存储出现在索引中的 term 的位置信息
Payloads	.pay	存储额外的 per-position 元数据信息，例如，字符偏移和用户有效负载
Norms	.nvd、.nvm	编码以后的 docs 和 fields 的长度及权重因子
Per-Document Values	.dvd、.dvm	编码后的附加评分因子或其他 per-document 信息
Term Vector Index	.tvx	将偏移存储到文档数据文件中
Term Vector Data	.tvd	保存 term vector 数据
Live Documents	.liv	活跃文档信息
Point values	.dii、.dim	保存索引点

下面我们开始分析三种类型节点各自的执行流程。

13.4.2 协调节点流程

协调节点负责解析请求，将请求转发给主节点。

处理线程：http_server_worker。

协调节点注册的 REST action 为 create_snapshot_action，相应的 Handler 为 RestCreateSnapshotAction 类。当协调节点收到客户端请求后，在 BaseRestHandler#handleRequest 中处理请求，调用 RestCreateSnapshotAction#prepareRequest 解析 REST 请求，将请求封装为 CreateSnapshotRequest 结构，然后将该请求发送到 Master 节点。

```
public RestChannelConsumer prepareRequest(final RestRequest request, final
NodeClient client) throws IOException {
```

```

//将请求封装为 CreateSnapshotRequest 结构
CreateSnapshotRequest createSnapshotRequest = createSnapshotRequest
(request.param("repository"), request.param("snapshot"));
request.applyContentParser(p -> createSnapshotRequest.source
(p.mapOrdered()));
//设置超时、wait_for_completion 等参数信息
createSnapshotRequest.masterNodeTimeout(request.paramAsTime
("master_timeout", createSnapshotRequest.masterNodeTimeout()));
createSnapshotRequest.waitForCompletion(request.paramAsBoolean
("wait_for_completion", false));
return channel -> client.admin().cluster().createSnapshot
(createSnapshotRequest, new RestToXContentListener<>(channel));
}

```

在 `TransportMasterNodeAction.AsyncSingleAction#doStart` 方法中，判断本地是否是主节点，如果是主节点，则转移到 snapshot 线程处理，否则发送 action 为 `cluster:admin/snapshot/create` 的 RPC 请求到主节点，request 为组装好的 `CreateSnapshotRequest` 结构。

代码摘要如下：

```

protected void doStart(ClusterState clusterState) {
    if (nodes.isLocalNodeElectedMaster() || localExecute(request)) {
        //本地是主节点，在新的线程池中处理请求
        threadPool.executor(executor).execute(new ActionRunnable(delegate) {
            protected void doRun() throws Exception {
                masterOperation(task, request, clusterState, delegate);
            }
        });
    } else { //转发到主节点。request 为组装好的 CreateSnapshotRequest 结构
        transportService.sendRequest(masterNode, actionName, request, new
        ActionListenerResponseHandler<Response>(listener,
            TransportMasterNodeAction.this::newResponse)
        );
    }
}

```

从实现角度来说，协调节点和主节点都会执行 `TransportMasterNodeAction.AsyncSingleAction#doStart` 方法，只是调用链不同。

13.4.3 主节点流程

主节点的主要处理过程是将请求转换成内部需要的数据结构，提交一个集群任务进行处理，集群任务处理后生成的集群状态中会包含请求快照的信息，主节点将新生成的集群状态广播下去，数据节点收到后执行相应的实际数据的快照处理。

执行本流程的线程池：`http_server_worker->snapshot->masterService#updateTask`。

如上一节所述，主节点收到协调节点发来的请求也是在 `TransportMasterNodeAction.AsyncSingleAction#doStart` 方法中处理的，在 `snapshot` 线程池中执行 `TransportCreateSnapshotAction#masterOperation` 方法。将收到的 `CreateSnapshotRequest` 请求转换成 `SnapshotsService.SnapshotRequest` 结构，调用 `snapshotsService.createSnapshot` 方法提交一个集群任务。

```
protected void masterOperation(...) {
    snapshotsService.createSnapshot(snapshotRequest, new SnapshotsService.
CreateSnapshotListener() {
        public void onResponse() {
            //根据需要注册一个 Listener，快照执行完毕才返回响应给客户端
            if (request.waitForCompletion()) {
                snapshotsService.addListener(new SnapshotsService.
SnapshotCompletionListener() {
                    public void onSnapshotCompletion(Snapshot snapshot,
SnapshotInfo snapshotInfo) {
                        ...
                    }

                    public void onSnapshotFailure(Snapshot snapshot, Exception e) {
                        ...
                    }
                });
            } else { //给客户端返回结果，快照任务后台执行
                listener.onResponse(new CreateSnapshotResponse());
            }
        }

        //对执行失败的处理
        public void onFailure(Exception e) {
```

```

        listener.onFailure(e);
    }
});

```

我们忽略对请求的验证和超时处理，以及失败处理等不重要的细节，摘取 snapshotsService.createSnapshot 的主要实现如下：

```

public void createSnapshot(final SnapshotRequest request, final
CreateSnapshotListener listener) {
    clusterService.submitStateUpdateTask(request.cause(), new
ClusterStateUpdateTask() {
        //定义要执行的任务
        public ClusterState execute(ClusterState currentState) {
            //快照任务不能并行，同一时间只能有一个快照在执行
            if (snapshots == null || snapshots.entries().isEmpty()) {
                //要快照的索引列表
                List<IndexId> snapshotIndices = repositoryData.
resolveNewIndices(indices);
                newSnapshot = new SnapshotsInProgress.Entry(new Snapshot
(repositoryName, snapshotId),
                    request.includeGlobalState(),
                    request.partial(),
                    State.INIT, //初始状态为 INIT
                    snapshotIndices,
                    System.currentTimeMillis(),
                    repositoryData.getGenId(),
                    null);
                snapshots = new SnapshotsInProgress(newSnapshot);
            } else {
                throw new ConcurrentSnapshotExecutionException(repositoryName,
snapshotName, " a snapshot is already running");
            }
            //根据 snapshots 信息生成新的集群状态
            return ClusterState.builder(currentState).putCustom
(SnapshotsInProgress.TYPE, snapshots).build();
        }
    }
}

```

//待 State 为 INIT 集群状态处理成功后，发送 State 为 START 的集群状态

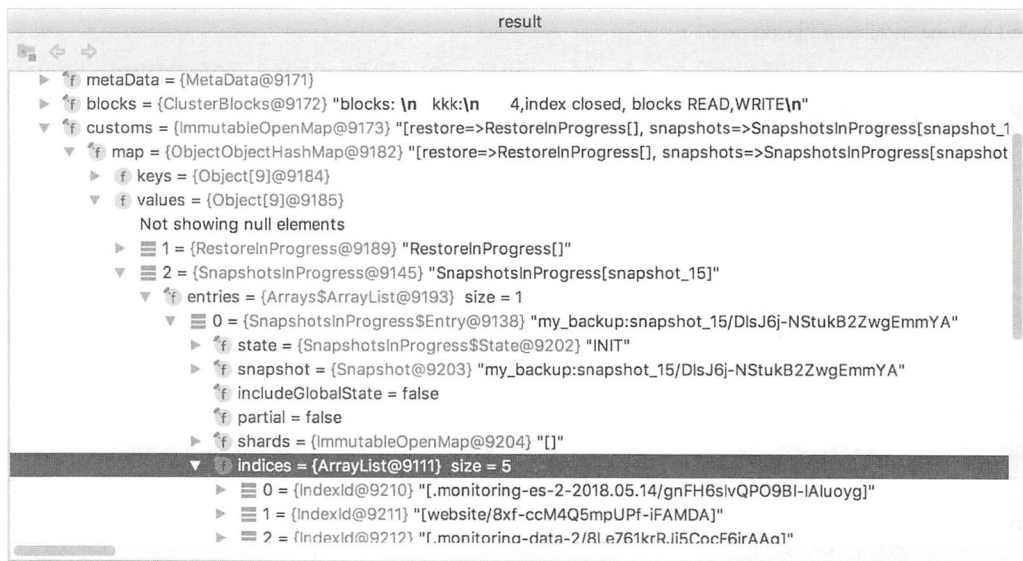

```

        public void clusterStateProcessed(String source, ClusterState
oldState, final ClusterState newState) {
            if (newSnapshot != null) {
                threadPool.executor(ThreadPool.Names.SNAPSHOT).execute(() ->
                    beginSnapshot(newState, newSnapshot, request.partial(),
listener)

                );
            }
        }
    }
}

```

提交的任务在 `masterService#updateTask` 线程中执行。在任务中对请求做非法检查，以及是否已经有快照在执行等验证操作，然后将快照请求的相关信息放入集群状态中，广播到集群的所有节点，触发数据节点对实际数据的处理。快照信息在集群状态的 `customs` 字段中，其组织结构如下图所示。



主节点控制数据节点执行快照的方式，就是通过把要执行的快照命令放到集群信息中广播下去。在执行快照过程中，主节点分成两个步骤，下发两次集群状态。首次发送时，快照信息中的 `State` 设置为 `INIT`，数据节点进行一些初始化操作。待数据节点将这个集群状态处理完毕后，主节点准备下发第二次集群状态。

第二次集群状态在 `SnapshotsService#beginSnapshot` 方法中构建。

在下发第二次集群状态前，主节点会先将全局元信息和索引的元数据信息写入仓库。

```
public void initializeSnapshot (SnapshotId snapshotId, List<IndexId> indices,
    Metadata clusterMetadata) {
    try {
        // 写全局元信息
        globalMetadataFormat.write(clusterMetadata, snapshotsBlobContainer,
            snapshotId.getUUID());

        // 为快照的每个索引写索引级元信息
        for (IndexId index : indices) {
            final IndexMetadata indexMetadata = clusterMetadata.index
(index.getName());
            final BlobPath indexPath = basePath().add("indices").add
(index.getId());
            final BlobContainer indexMetadataBlobContainer = blobStore()
.blobContainer(indexPath);
            indexMetadataFormat.write(indexMetadata, indexMetadataBlobContainer,
            snapshotId.getUUID());
        }
    } catch (IOException ex) {
        throw new SnapshotCreationException(metadata.name(), snapshotId, ex);
    }
}
```

在新的集群状态中，将 State 设置为 STARTED，并且根据将要快照的索引列表计算出分片列表（注意全是主分片），数据节点收到后真正开始执行快照。

13.4.4 数据节点流程

数据节点负责实际的快照实现，从全部将要快照的分片列表找出存储于本节点的分片，对这些分片创建 Lucene 快照，复制文件。

1. 对 ClusterState 的处理

对收到的集群状态的处理在 clusterApplierService#updateTask 线程池中。启动快照时，在 snapshot 线程池中。

数据节点对主节点发布的集群状态（ClusterState）的统一处理在 ClusterApplierService#

callClusterStateListeners 方法中，clusterStateListeners 中存储了所有需要对集群状态进行处理的模块。当收到集群状态时，遍历这个列表，调用各个模块相应的处理函数。快照模块对此的处理在 SnapshotShardsService#clusterChanged 方法中。在该方法中，在做完一些简单的验证之后，调用 SnapshotShardsService#processIndexShardSnapshots 进入主要处理逻辑。

数据节点对第一次集群状态的处理实际上没做什么有意义的操作。

对第二次集群状态的处理是真正快照的核心实现。

主节点第二次下发的集群状态中包含了要进行快照的分片列表。数据节点收到后过滤一下本地有哪些分片，构建一个新的列表，后续要进行快照的分片就在这个列表中。

```
for (ObjectObjectCursor<ShardId, ShardSnapshotStatus> shard :
entry.shards()) {
    // 准备本节点要处理的分片
    if (localNodeId.equals(shard.value.nodeId())) {
        if (shard.value.state() == State.INIT && (snapshotShards == null
|| !snapshotShards.shards.containsKey(shard.key))) {
            logger.trace("{} - Adding shard to the queue", shard.key);
            startedShards.put(shard.key, new IndexShardSnapshotStatus());
        }
    }
}
//本节点要处理的分片列表
newSnapshots.put(entry.snapshot(), startedShards);
```

然后遍历本地要处理的分片列表，在 snapshot 线程池中对分片并行执行快照处理。并行数量取决于 snapshot 线程池中的线程个数，默认的线程数最大值为：

```
min(5, (处理器数量)/2)
```

处理完毕后，向主节点发送 RPC 请求以更新相应分片的快照状态。

```
if (newSnapshots.isEmpty() == false) {
    Executor executor = threadPool.executor(ThreadPool.Names.SNAPSHOT);
    for (final Map.Entry<Snapshot, Map<ShardId, IndexShardSnapshotStatus>>
entry : newSnapshots.entrySet()) {
        //shard 级并发执行快照
        for (final Map.Entry<ShardId, IndexShardSnapshotStatus> shardEntry :
entry.getValue().entrySet()) {
```

```
executor.execute(new AbstractRunnable() {
    public void doRun() {
        //对特定分片执行快照
        snapshot(indexShard, snapshot, indexId, shardEntry.getValue());
    }

    public void onAfter() {
        //向 Master 节点发送请求以更新快照状态
        final Exception exception = failure.get();
        if (exception != null) {
            final String failure = ExceptionsHelper.detailedMessage
(exception);

            notifyFailedSnapshotShard(snapshot, shardId, localNodeId,
failure, masterNode);
        } else {
            notifySuccessfulSnapshotShard(snapshot, shardId,
localNodeId, masterNode);
        }
    }
});
}
```

2. 对一个特定分片的快照实现

现在我们讨论数据节点对单个分片执行快照的过程，所有的分片执行一样的流程。本节以单个分片的处理为背景。

Lucene 快照

由于 ES 的快照基于 Lucene 快照实现，本节我们先介绍 Lucene 快照的实现原理。

Lucene 的快照在 `SnapshotDeletionPolicy#snapshot` 方法中实现。该方法返回一个提交点，通过提交点可以获取分片的最新状态，包括全部 Lucene 分段文件的列表。从得到这个列表开始，列表中的文件都不会被删除，直到释放提交点。

先看一下 Lucene 快照的实现：

```
IndexCommitRef(SnapshotDeletionPolicy deletionPolicy) throws IOException {
    //调用 Lucene 接口创建快照，返回提交点
```

```

    indexCommit = deletionPolicy.snapshot();
    //处理完毕, 释放快照资源
    onClose = () -> deletionPolicy.release(indexCommit);
}

```

`deletionPolicy` 初始化时使用的是 `KeepOnlyLastCommitDeletionPolicy`, 对最后一次提交进行快照, 包含全部 Lucene 分段, 代表 Lucene 索引在磁盘中的最新状态。

```

this.deletionPolicy = new CombinedDeletionPolicy(
    new SnapshotDeletionPolicy(new KeepOnlyLastCommitDeletionPolicy()), ...);

```

ES 快照整体过程

数据节点在 `snapshot` 线程池中执行 `SnapshotShardsService#snapshot` 方法, 对特定分片创建快照。

在做完一些验证工作后, 调用 Lucene 接口创建快照, 返回 `Engine.IndexCommitRef`, 其中含有最重要的提交点。然后根据 Lucene 提交点创建 ES 快照:

```

//创建 Lucene 快照, 快照前先执行 flush 操作, 但不执行 refresh 操作
try (Engine.IndexCommitRef snapshotRef = indexShard.acquireIndexCommit(true)) {
    //根据 Lucene 提交点创建快照
    repository.snapshotShard(indexShard, snapshot.getSnapshotId(),
indexId, snapshotRef.getIndexCommit(), snapshotStatus);
}

```

我们通过一个例子来分析整个过程, 为了简化场景, 我们在单节点的集群上创建了只有一个主分片的索引 `website`。写入数据, 执行两次提交, 每次提交后创建一个 ES 快照。在第二次执行快照 `snapshot_2` 时, ES 数据磁盘中存储的数据文件如下:

```

tree indices/IGETEzgOSC6BeFAnGw6vDg
indices/IGETEzgOSC6BeFAnGw6vDg
├── 0
│   ├── _state
│   │   └── state-0.st
│   ├── index
│   │   ├── _0.cfe
│   │   ├── _0.cfs
│   │   └── _0.si

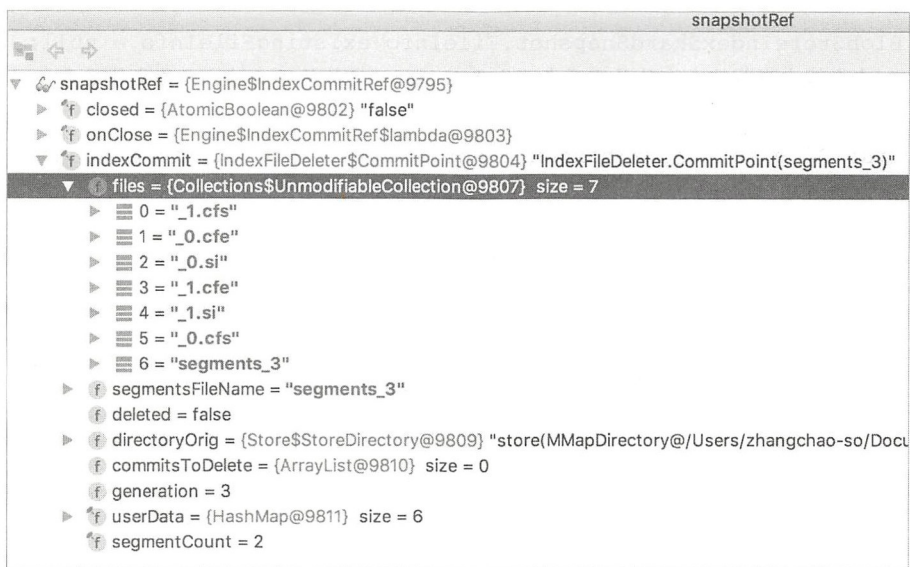
```



```
| | | _1.cfe
| | | _1.cfs
| | | _1.si
| | | segments_2
| | | write.lock
| | └─ translog
| |   | └─ translog-1.ckp
| |   | └─ translog-1.tlog
| |   | └─ translog-2.tlog
| |   └─ translog.ckp
└─ _state
    └─ state-2.st
```

可以看到 Lucene 文件中存在两个分段, 分别是 `_0.*` 和 `_1.*`。其中, 分段 0 有三个文件: `_0.cfe`、`_0.cfs`、`_0.si`。

获取 Lucene 快照后, 返回值 `snapshotRef` 中包含提交点 (indexCommit) 信息, 其中 Lucene 的文件列表如下图所示。



files 中包含全部两个 Lucene 分段文件。接下来处理这个 Lucene 提交点。

ES 快照的核心流程就是根据这个提交点创建快照。封装的方法为 `BlobStoreRepository.snapshotContext#snapshot`。下面介绍主要实现过程。

根据 Lucene 提交点计算两个列表

当前的 Lucene 提交点代表分片的最新状态，它包含全部 Lucene 分段。如果不考虑增量备份，则把这个文件列表全部复制到仓库就可以了。但是我们要实现的是每次快照都是增量的。实现方法就是计算出两个列表：

(1) 新增文件列表，代表将要复制到仓库的文件。遍历 Lucene 提交点中的文件列表，如果仓库中已存在，则过滤掉，得到一个新增文件列表。

(2) 当前快照使用的全部文件列表，未来通过它找到某个快照相关的全部相关文件。这个列表的内容就是 Lucene 提交点中的文件列表的全部文件。

计算两个文件列表的过程如下:

```

fileNames = snapshotIndexCommit.getFileNames();
//遍历 Lucene 提交点中的文件列表
for (String fileName : fileNames) {
    //检查是否需要中止快照
    if (snapshotStatus.aborted()) {
        throw new IndexShardSnapshotFailedException(shardId, "Aborted");
    }
    BlobStoreIndexShardSnapshot.FileInfo existingFileInfo = null;
    //在仓库中查找这个文件是否存在，由于 Lucene 文件的不变性，只要文件名存在，文件就是
    //相同的，不检测校验和
    List<BlobStoreIndexShardSnapshot.FileInfo> filesInfo =
snapshots.findPhysicalIndexFiles(fileName);
    if (filesInfo != null) {
        for (BlobStoreIndexShardSnapshot.FileInfo fileInfo : filesInfo) {
            if (fileInfo.isSame(md) && snapshotFileExistsInBlobs(fileInfo, blobs)) {
                //仓库中已存在
                existingFileInfo = fileInfo;
                break;
            }
        }
    }
    if (existingFileInfo == null) {
        BlobStoreIndexShardSnapshot.FileInfo snapshotFileInfo = new
BlobStoreIndexShardSnapshot.FileInfo(...);
        indexCommitPointFiles.add(snapshotFileInfo); //添加到本次快照的全部文
        //件列表
    }
}

```

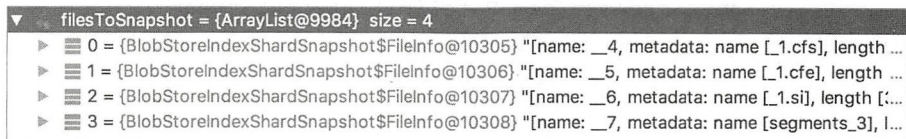
```

        filesToSnapshot.add(snapshotFileInfo); // 添加到新增文件列表
    } else {
        indexCommitPointFiles.add(existingFileInfo); // 添加到本次快照的全部文
                                                    // 件列表
    }
}

```

在此我们需要介绍 Lucene 文件的“不变性”，除了 write.lock、segments.gen 两个文件，其他所有文件都不会更新，只写一次（write once）。锁文件 write.lock 不需要复制。segments.gen 是较早期的 Lucene 版本中存在的一种文件，我们不再讨论。因此，所有需要复制的文件都是不变的，无须考虑被更新的可能。所以在增量备份时，通过文件名就可以识别唯一的文件。但是在存储到仓库时，ES 将文件全部重命名为以一个递增序号为名字的文件，并维护了对应关系。

在本例中，两个列表计算完的结果如下图所示。



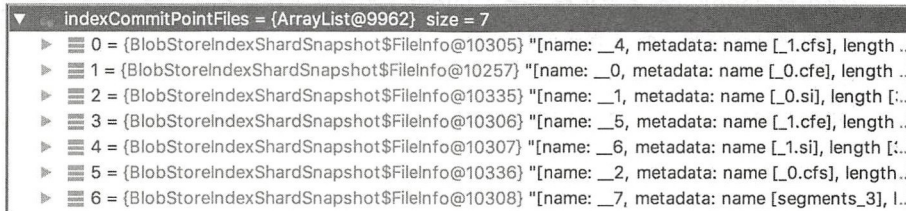
```

▼ filesToSnapshot = {ArrayList@9984} size = 4
  ▶ 0 = {BlobStoreIndexShardSnapshot$FileInfo@10305} "[name: __4, metadata: name [_1.cfs], length ...
  ▶ 1 = {BlobStoreIndexShardSnapshot$FileInfo@10306} "[name: __5, metadata: name [_1.cfe], length ...
  ▶ 2 = {BlobStoreIndexShardSnapshot$FileInfo@10307} "[name: __6, metadata: name [_1.si], length [...
  ▶ 3 = {BlobStoreIndexShardSnapshot$FileInfo@10308} "[name: __7, metadata: name [segments_3], l...

```

新增文件列表 filesToSnapshot 只包含从上一次快照之后到当前的新增文件。以第一条为例，__4 为复制到仓库时的目标文件名，_1.cfs 为源文件名。

完整文件列表 indexCommitPointFiles 包含分片的全部 Lucene 分段和最后的提交点文件（segments_3），如下图所示。



```

▼ indexCommitPointFiles = {ArrayList@9962} size = 7
  ▶ 0 = {BlobStoreIndexShardSnapshot$FileInfo@10305} "[name: __4, metadata: name [_1.cfs], length ...
  ▶ 1 = {BlobStoreIndexShardSnapshot$FileInfo@10257} "[name: __0, metadata: name [_0.cfe], length ...
  ▶ 2 = {BlobStoreIndexShardSnapshot$FileInfo@10335} "[name: __1, metadata: name [_0.si], length [...
  ▶ 3 = {BlobStoreIndexShardSnapshot$FileInfo@10306} "[name: __5, metadata: name [_1.cfe], length ...
  ▶ 4 = {BlobStoreIndexShardSnapshot$FileInfo@10307} "[name: __6, metadata: name [_1.si], length [...
  ▶ 5 = {BlobStoreIndexShardSnapshot$FileInfo@10336} "[name: __2, metadata: name [_0.cfs], length...
  ▶ 6 = {BlobStoreIndexShardSnapshot$FileInfo@10308} "[name: __7, metadata: name [segments_3], l...

```

复制 Lucene 物理文件

在开始复制之前，快照任务被设置为 STARTED 阶段。

现在开始复制新增文件，遍历新增文件列表，将这些文件复制到仓库：

```

for (BlobStoreIndexShardSnapshot.FileInfo snapshotFileInfo : filesToSnapshot) {
    snapshotFile(snapshotFileInfo);
}

```

复制过程中实现限速，并且计算校验和。Lucene 每个文件的元信息中有计算好的校验和，在数据复制过程中，一边复制，一边计算，复制完毕后对比校验和是否相同，以验证复制结果是否正确。校验和很重要，在手工备份数据时，复制完毕后我们实际上不知道复制的数据是否正确。校验和是对数据进行整型加法的计算，不会消耗多少 CPU 资源。

```
private void snapshotFile(final BlobStoreIndexShardSnapshot.FileInfo
fileInfo) throws IOException {
    final String file = fileInfo.physicalName();
    try (IndexInput indexInput = store.openVerifyingInput(file,
IOContext.READONCE, fileInfo.metadata())) {
        for (int i = 0; i < fileInfo.numberOfParts(); i++) {
            InputStream inputStream = inputStreamIndexInput;
            //初始化限速模块
            if (snapshotRateLimiter != null) {
                inputStream = new RateLimitingInputStream
(inputStreamIndexInput, snapshotRateLimiter,
                    snapshotRateLimitingTimeInNanos::inc);
            }
            inputStream = new AbortableInputStream(inputStream,
fileInfo.physicalName());
            //复制文件
            blobContainer.writeBlob(fileInfo.partName(i), inputStream, partBytes);
        }
        //比较 checksum 是否正确。一是 Lucene 元信息中存储的校验和，二是数据复制到目
        //的地址过程中，计算出来的校验和
        Store.verify(indexInput);
        snapshotStatus.addProcessedFile(fileInfo.length());
    }
}
```

复制文件的 `blobContainer.writeBlob` 是一个虚方法，对于不同的仓库文件系统有不同的实现，对于共享文件系统(fs)来说，复制过程通过 `Streams.copy` 实现，并在复制完成后执行 `IOUtils.fsync` 刷盘。

生成快照文件

这个文件是快照的描述信息，包含本次快照的相关描述，以及与其相关的 Lucene 文件。

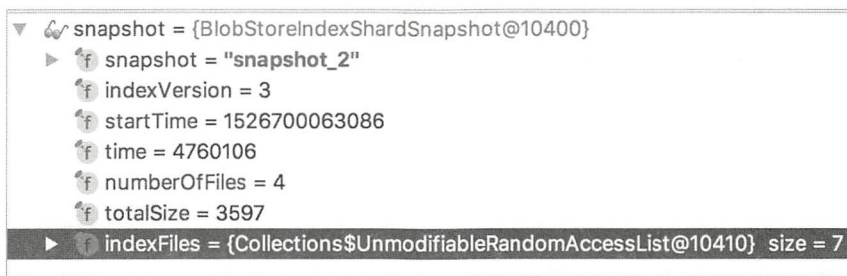
```
BlobStoreIndexShardSnapshot snapshot = new BlobStoreIndexShardSnapshot
```



```
(snapshotId.getName(),
    snapshotIndexCommit.getGeneration(), indexCommitPointFiles,
    snapshotStatus.startTime(),
    System.currentTimeMillis() - snapshotStatus.startTime(),
    indexNumberOfFiles, indexTotalFilesSize);

//写快照文件, 快照文件中描述了本次快照要使用的所有相关 Lucene 分段
indexShardSnapshotFormat.write(snapshot, blobContainer,
    snapshotId.getUUID());
```

在本例中, 快照文件内容如下图所示。



文件列表代表分片的最新状态, 包括分片的全部 Lucene 分段。

删除快照无关文件

遍历仓库中的全部快照, 包括刚刚执行完的快照, 删除仓库中不被任何快照关联的文件。

BlobStoreRepository.Context#finalize 方法负责找出这些文件并删除。

```
protected void finalize(List<SnapshotFiles> snapshots, int fileListGeneration,
    Map<String, BlobMetaData> blobs) {
    BlobStoreIndexShardSnapshots newSnapshots = new
    BlobStoreIndexShardSnapshots(snapshots);
    //从当前分片的快照中删除索引 index-*, 这个 index-* 文件是对快照列表的索引
    //blobName 是仓库中分片下一个具体的文件, 例如, __1、index-*, snap-*
    for (String blobName : blobs.keySet()) {
        if (indexShardSnapshotsFormat.isTempBlobName(blobName) ||
        blobName.startsWith (SNAPSHOT_INDEX_PREFIX)) {
            blobContainer.deleteBlob(blobName);
        }
    }
}
```



```
//遍历当前分片仓库中的数据文件，删除快照中未使用的文件
for (String blobName : blobs.keySet()) {
    //删除无用的文件，数据 BLOB 前缀 DATA_BLOB_PREFIX 值为__
    if (blobName.startsWith(DATA_BLOB_PREFIX)) {
        if (newSnapshots.findNameFile(BlobStoreIndexShardSnapshot.
FileInfo.canonicalName(blobName)) == null) {
            blobContainer.deleteBlob(blobName);
        }
    }
}

//如果全部快照都删除，则不创建 index-*索引文件
if (snapshots.size() > 0) {
    //写 index-*索引文件
    indexShardSnapshotsFormat.writeAtomic(newSnapshots, blobContainer,
Integer.toString(fileListGeneration));
}
}
```

此处传入 `finalize` 方法的第一个参数是该分片的全部快照的列表，然后做了三件事：

- 删除旧的 `index-*`索引文件；
- 删除仓库中不被快照使用的文件，快照列表由函数第一个参数传入，值为分片的全部快照；
- 创建新的 `index-*`索引文件。

`finalize` 方法执行完成后，快照任务被设置为 `DONE` 阶段。

思考一下，在 `finalize` 方法的参数中，传入的快照列表如果不是全部快照，而是其中一部分，则会发生什么？

回顾

简单回顾一下两次快照的过程。第一次创建快照时，`snapshot_1` 的文件列表如下表所示。

Lucene 文件	复制到仓库后的文件
<code>_0.cfe</code>	<code>__0</code>
<code>_0.cfs</code>	<code>__1</code>
<code>_0.si</code>	<code>__2</code>
<code>segments_1</code>	<code>__3</code>

第二次创建快照时，snapshot_2 的文件列表如下表左列所示。segments_1 已删除，不属于 snapshot_2，只有加粗的文件是 snapshot_2 需要复制的新增内容。

Lucene 文件	仓库中的文件
<u>_0.cfe</u>	<u>_0</u>
<u>_0.cfs</u>	<u>_1</u>
<u>_0.si</u>	<u>_2</u>
segments_1	<u>_3</u>
<u>_1.cfe</u>	<u>_4</u>
<u>_1.cfs</u>	<u>_5</u>
<u>_1.si</u>	<u>_6</u>
segments_2	<u>_7</u>

第二次快照文件执行完毕后，仓库中的文件结构如下所示。

```
tree indices
indices
├── VFJcUdFxQjm3Q0vpcTqnRQ
│   ├── 0
│   │   ├── __0
│   │   ├── __1
│   │   ├── __2
│   │   ├── __3
│   │   ├── __4
│   │   ├── __5
│   │   ├── __6
│   │   ├── __7
│   │   ├── index-0
│   │   ├── snap-4OphcIKTSbubAY-SlBzQDw.dat
│   │   └── snap-UTSw17N1QkG3gHoA8EN2Rw.dat
│   ├── meta-4OphcIKTSbubAY-SlBzQDw.dat
│   └── meta-UTSw17N1QkG3gHoA8EN2Rw.dat
```

仓库中有三种类型的文件：

- 以__为前缀的是数据文件，是 Lucene 中的分段（segment）文件被重命名后的文件。
- snap-*.dat 为快照文件，描述了快照名称，与快照相关的数据文件（使用的文件列表）等信息。

- `index-*`中描述了当前分片所有的快照信息，是快照列表的索引文件。
- `meta-*`为索引的元数据信息，仓库根目录下也有 `meta-*`文件，根目录下的是全局集群元信息。

13.5 删除快照实现原理

ES 删除快照的核心思想就是，在要删除的快照所引用的物理文件中，对不被任何其他快照使用的文件执行删除。每个快照都在自己的元信息文件（`snap-*`）中描述了本快照使用的文件列表。想要删除一些文件时，也不需要引用计数，只要待删除文件不被其他快照使用就可以安全删除。

快照删除过程/取消过程涉及 3 种类型的节点：

- **协调节点**，接收客户端请求、转发到主节点。
- **主节点**，将删除创建快照相关的请求信息放到集群状态中广播下去，删除快照和取消运行中的快照是同一个请求。数据节点负责取消运行中的快照创建任务，主节点负责删除已创建完毕的快照。无论如何，集群状态都会广播下去。当集群状态发布完毕，主节点开始执行删除操作。所以现在知道为什么主节点也要访问仓库了。删除操作确实没有必要要求各个数据节点去执行，任何节点都能看到仓库的全部数据，只需要单一节点执行删除即可，因此删除操作由主节点执行。
- **数据节点**，负责取消正在运行的快照任务。

13.5.1 协调节点流程

协调节点的任务与创建快照时相同，负责协调节点负责解析请求，将请求转发给主节点。

处理线程：`http_server_worker`。

删除快照的相应的 REST action 为 `delete_snapshot_action`。注册的 Handler 为 `RestDeleteSnapshotAction`。收到删除快照的 REST 请求后，同样在 `BaseRestHandler#handleRequest` 中进行处理，然后调用 `RestDeleteSnapshotAction#prepareRequest` 解析 REST 请求，将请求封装为 `DeleteSnapshotRequest` 结构，然后将该请求发送到 Master 节点。

由于处理过程与创建快照时类似，我们省略相关代码的引用。

同样在 `TransportMasterNodeAction.AsyncSingleAction#doStart` 方法中判断本地是否是主节点，如果是主节点，则本地在 `snapshot` 线程池中执行，否则将转发过去，请求的 action 为 `cluster:admin/snapshot/delete`。



13.5.2 主节点流程

主节点收到协调节点的请求后提交集群任务，将请求信息放到新的集群状态中广播下去，数据节点收到后检查是否有运行中的快照任务需要取消，如果没有，则不做其他操作。主节点的集群状态发布成功后，执行快照删除操作。

执行本流程的线程池：`http_server_worker->generic->masterService#updateTask->snapshot`。

主节点收到协调节点发来的请求也是在 `TransportMasterNodeAction.AsyncSingleAction#doStart` 方法中处理的，在 `generic` 线程池中执行 `TransportDeleteSnapshotAction#masterOperation`，接着调用 `SnapshotsService#deleteSnapshot` 提交集群任务。

1. 提交集群任务

将删除快照请求信息放到集群状态中，当集群状态发布成功后，执行删除快照逻辑。

```
private void deleteSnapshot(...) {
    //提交集群任务
    clusterService.submitStateUpdateTask("delete snapshot", new
ClusterStateUpdateTask(priority) {
        //将删除快照的请求信息放到新的集群状态中
        public ClusterState execute(ClusterState currentState) throws
Exception {
            SnapshotsInProgress snapshots = currentState.custom
(SnapshotsInProgress.TYPE);
            SnapshotsInProgress.Entry snapshotEntry = snapshots != null ?
snapshots.snapshot(snapshot) : null;
            if (snapshotEntry == null) {
                //快照没有运行，将删除请求信息放到 Custom 字段的
                //SnapshotDeletionsInProgress 结构中
                clusterStateBuilder.putCustom(SnapshotDeletionsInProgress.TYPE,
deletionsInProgress);
            }
            else{
                //将删除快照请求的相关信息放到 Customs 的 SnapshotsInProgress 中，
                //并将 state 设置为 ABORTED
                SnapshotsInProgress.Entry newSnapshot = new
SnapshotsInProgress.Entry(snapshotEntry, State.ABORTED, shards);
                snapshots = new SnapshotsInProgress(newSnapshot);
```



```
        clusterStateBuilder.putCustom(SnapshotsInProgress.TYPE,
snapshots);
    }

    snapshots = new SnapshotsInProgress(newSnapshot);
    clusterStateBuilder.putCustom(SnapshotsInProgress.TYPE, snapshots);

    return clusterStateBuilder.build();
}

//处理集群状态发布失败的情况
public void onFailure(String source, Exception e) {
}

//集群状态发布完毕
public void clusterStateProcessed(String source, ClusterState
oldState, ClusterState newState) {

    if (waitForSnapshot) {
        //处理客户端等待完成的情况
    } else {
        //删除快照文件
        deleteSnapshotFromRepository(snapshot, listener, repositoryStateId);
    }
}

});
}
```

主节点会判断要删除的快照是正在进行中的，还是已完成的，对进行中的快照执行取消逻辑，对已完成的快照执行删除逻辑，构建出的集群状态是不同的。

对于删除过程，下发的集群状态内容如下图所示。删除请求信息放在 `customs` 的 `SnapshotDeletionsInProgress` 字段中。



```
result = {ClusterState@9103} "cluster uuid: olrqNUxhTC2OVVG8KyXJ_w\nversion: 8\ninstance uid: GX8QvIWZTymc_ICc9BM1
  version = 8
  stateUUID = "GX8QvIWZTymc_ICc9BM1dQ"
  routingTable = {RoutingTable@9080} "routing_table (version 4):\n-- index [[website/VpwFLePUT1ynJ5CQfOx7qA]]\n---
  nodes = {DiscoveryNodes@9027} "nodes: \n {V6xmvW}{V6xmvWyTm-vlY-bqp-cVg}{ceOpsINRR9mAI2SMiJhkdw}{127.0
  metaData = {MetaData@9076}
  blocks = {ClusterBlocks@9077} ""
  customs = {ImmutableOpenMap@9109} "[snapshot_deletions=>SnapshotDeletionsInProgress[snapshot_2], restore=>Re
    map = {ObjectObjectHashMap@9121} "[snapshot_deletions=>SnapshotDeletionsInProgress[snapshot_2], restore=>Re
      keys = {Object[9]@9123}
      values = {Object[9]@9124}
      Not showing null elements
    0 = {SnapshotDeletionsInProgress@9067} "SnapshotDeletionsInProgress[snapshot_2]"
      entries = {Collections$UnmodifiableRandomAccessList@9129} size = 1
        0 = {SnapshotDeletionsInProgress$Entry@9188}
          snapshot = {Snapshot@9061} "my_backup:snapshot_2/bvEfj_HUR82ztwzM-sX9-g"
            repository = "my_backup"
            snapshotId = {SnapshotId@9190} "snapshot_2/bvEfj_HUR82ztwzM-sX9-g"
            hashCode = 608580754
            startTime = 1526953709974
            repositoryStateId = 7
          1 = {RestoreInProgress@9068} "RestoreInProgress[]"
          4 = {SnapshotsInProgress@8961} "SnapshotsInProgress[]"
          keyMixer = -1655422024
```

对于取消过程，下发的集群状态内容如下图所示。删除请求信息放在 customs 的 SnapshotsInProgress 字段中，并将 State 设置为 ABORTED。创建快照时也放在 SnapshotsInProgress 字段中，区别就是创建快照时 State 为 STARTED。

```
event = {ClusterChangedEvent@9125}
  source = "apply cluster state (from master [master {fc6s0S0}{fc6s0S0hRi2yJvMo54qt_g}{OJWZgJGwSp28m_A
  previousState = {ClusterState@9132} "cluster uuid: olrqNUxhTC2OVVG8KyXJ_w\nversion: 15\ninstance uid: 3mcYRv7M
  state = {ClusterState@9133} "cluster uuid: olrqNUxhTC2OVVG8KyXJ_w\nversion: 16\ninstance uid: 0ikesn6mQVioqM2s
    version = 16
    stateUUID = "0ikesn6mQVioqM2sH_ODHA"
    routingTable = {RoutingTable@9138} "routing_table (version 7):\n-- index [[website/VpwFLePUT1ynJ5CQfOx7qA]]
    nodes = {DiscoveryNodes@8954} "nodes: \n {fc6s0S0}{fc6s0S0hRi2yJvMo54qt_g}{OJWZgJGwSp28m_AfAiTmlw
    metaData = {MetaData@9139}
    blocks = {ClusterBlocks@9140} ""
    customs = {ImmutableOpenMap@9141} "[snapshot_deletions=>SnapshotDeletionsInProgress[], snapshots=>Sna
      map = {ObjectObjectHashMap@9150} "[snapshot_deletions=>SnapshotDeletionsInProgress[], snapshots=>Sna
        keys = {Object[9]@9152}
        values = {Object[9]@9153}
        Not showing null elements
        5 = {SnapshotDeletionsInProgress@9155} "SnapshotDeletionsInProgress[]"
        6 = {SnapshotsInProgress@9126} "SnapshotsInProgress[snapshot_3]"
          entries = {Arrays$ArrayList@9160} size = 1
            0 = {SnapshotsInProgress$Entry@9163} "my_backup:snapshot_3/HayjQ7cBQbS9YsLfzXVW3g"
              state = {SnapshotsInProgress$State@9165} "ABORTED"
              snapshot = {Snapshot@9166} "my_backup:snapshot_3/HayjQ7cBQbS9YsLfzXVW3g"
                includeGlobalState = true
                partial = false
              shards = {ImmutableOpenMap@9167} "[[website][1]=>ShardSnapshotStatus[state=ABORTED, r
              indices = {Collections$UnmodifiableRandomAccessList@9168} size = 1
              waitingIndices = {ImmutableOpenMap@9099} "[]"
              startTime = 1526955102050
              repositoryStateId = 9
            7 = {RestoreInProgress@9156} "RestoreInProgress[]"
          keyMixer = 1388753083
```

由于删除操作在主节点上执行，接下来我们进入主节点的快照删除过程。



2. 快照删除

主节点的集群状态发布完毕，`clusterStateProcessed` 方法负责发布成功后的处理逻辑。执行该方法的线程池为 `masterService#updateTask`，它调用 `SnapshotsService#deleteSnapshotFromRepository` 方法执行删除。该方法会转移到 `snapshot` 线程池执行具体的删除工作。

```
private void deleteSnapshotFromRepository(...) {
    threadPool.executor(ThreadPool.Names.SNAPSHOT).execute(() -> {
        //执行删除
        repository.deleteSnapshot(snapshot.getSnapshotId(), repositoryStateId);
        removeSnapshotDeletionFromClusterState(snapshot, null, listener);
    });
}
```

需要删除的内容包括元信息文件、索引分片，以及有可能要删除的整个索引目录，并且更新快照列表的 `index` 文件 (`index-*`)。主要删除逻辑如下：

```
public void deleteSnapshot(SnapshotId snapshotId, long repositoryStateId) {
    Metadata metaData = readSnapshotMetaData(snapshotId, snapshot.version(),
repositoryData.resolveIndices(indices), true);
    try {
        // 更新快照列表的索引文件 index-*, 将快照信息从快照列表中删除
        final RepositoryData updatedRepositoryData = repositoryData.
removeSnapshot(snapshotId);
        writeIndexGen(updatedRepositoryData, repositoryStateId);

        // 删除全局快照文件（仓库根目录下的）snap-*
        deleteSnapshotBlobIgnoringErrors(snapshot, snapshotId.getUUID());
        // 删除全局元信息文件 meta-*.dat
        deleteGlobalMetaDataBlobIgnoringErrors(snapshot, snapshotId.getUUID());
        // 删除全部索引
        for (String index : indices) {
            //删除索引的元信息文件 meta-*.dat
            indexMetaDataFormat.delete(indexMetaDataBlobContainer,
snapshotId.getUUID());
            if (metaData != null) {
                IndexMetaData indexMetaData = metaData.index(index);
                if (indexMetaData != null) {
                    for (int shardId = 0; shardId < indexMetaData.getNumberOfShards());
```



```
shardId++) {  
    //删除分片快照，这里是删除实际的 Lucene 文件  
    delete(snapshotId, snapshot.version(), indexId, new  
        ShardId(indexMetaData.getIndex(), shardId));  
}  
}  
}  
  
//删除仓库中已经不存在的索引目录  
for (final IndexId indexId : indicesToCleanUp) {  
    indicesBlobContainer.deleteBlob(indexId.getId());  
}  
}
```

快照是对每个分片创建的，如何删除分片快照是核心过程。接下来我们看一下如何删除分片的快照：

```
public void delete() {  
    Tuple<BlobStoreIndexShardSnapshots, Integer> tuple =  
        buildBlobStoreIndexShardSnapshots(blobs);  
    //仓库现有的快照列表  
    BlobStoreIndexShardSnapshots snapshots = tuple.v1();  
    int fileListGeneration = tuple.v2();  
  
    //删除分片的 snapshot 文件  
    indexShardSnapshotFormat(version).delete(blobContainer,  
        snapshotId.getUUID());  
  
    //构建需要保留的快照列表  
    List<SnapshotFiles> newSnapshotsList = new ArrayList<>();  
    for (SnapshotFiles point : snapshots) {  
        if (!point.snapshot().equals(snapshotId.getName())) {  
            newSnapshotsList.add(point);  
        }  
    }  
  
    //传入需要保留的快照列表，而非要删除的快照列表。仓库中存在的，
```



```
//但不被列表中快照使用的文件被删除
finalize(newSnapshotsList, fileListGeneration + 1, blobs);
}
```

调用创建快照时相同的 `finalize` 方法，调用该方法时传入一个快照列表，内部执行时遍历仓库中的文件，删除不被快照列表引用的文件。在创建快照时传入全部快照列表，在删除快照时，传入的是需要保留的快照列表。

以上一节创建快照中的例子为基础，我们删除 `snapshot_2`，看看哪些文件被删除。在删除 `snapshot_2` 之前，仓库中一共有 `snapshot_1`、`snapshot_2` 两个快照。

属于 `snapshot_1` 的文件有：

```
__0
__1
__2
__3
```

属于 `snapshot_2` 的文件有：

```
__0
__1
__2
__3
__4
__5
__6
__7
```

删除 `snapshot_2` 时，`finalize` 方法传入的快照列表为 `snapshot_1`，最终被删除的文件为 4、5、6、7，如下图所示。左列为该分片仓库中的现有文件，右列为 `snapshot_1` 的文件列表。

1	__0	1	__0
2	__1	2	__1
3	__2	3	__2
4	__3	4	__3
5	__4		
6	__5		
7	__6		
8	__7		
9		5	



3. 数据节点的取消过程

取消快照请求信息放在 `customs` 的 `SnapshotsInProgress` 字段中, `State` 为 `ABORTED`。数据节点对此的处理在 `SnapshotShardsService#processIndexShardSnapshots` 方法中。创建快照的主要过程也在这个方法中, 根据 `State` 状态判断需要启动或取消运行中的快照。

```
private void processIndexShardSnapshots(ClusterChangedEvent event) {
    if (snapshotsInProgress != null) {
        for (SnapshotsInProgress.Entry entry : snapshotsInProgress.entries()) {
            if (entry.state() == State.STARTED) {
                //处理快照创建
            } else if (entry.state() == State.ABORTED) {
                //处理快照取消
                if (snapshotShards != null) {
                    for (ObjectObjectCursor<ShardId, ShardSnapshotStatus>
shard : entry.shards()) {
                        if (snapshotStatus != null) {
                            switch (snapshotStatus.stage()) {
                                case INIT:
                                case STARTED:
                                    //设置中止标识
                                    snapshotStatus.abort();
                                case ...
                            }
                        }
                    }
                }
            }
        }
    }
}
```

取消快照的 `abort` 实现只是设置了中止标识, 运行中的快照会检查这个标识:

```
public void abort() {
    this.aborted = true;
}
```

快照运行过程中有多处会检查中止标识:



- 在计算需要复制的 Lucene 文件列表时;
- 在开始执行复制之前;
- 在数据开始复制数据之后的读取过程中。

由于运行中的快照大部分时间在执行数据复制, 因此取消操作大部分在读取数据时中断。

```
public int read(byte[] b, int off, int len) throws IOException {
    checkAborted();
    return in.read(b, off, len);
}

private void checkAborted() {
    if (snapshotStatus.aborted()) {
        throw new IndexShardSnapshotFailedException(shardId, "Aborted");
    }
}
```

运行中的快照被取消后, 复制到一半的快照数据文件由主节点负责清理。这个过程在主节点发布集群状态成功之后的快照删除逻辑中执行, 对一个已经取消的快照, 执行正常的快照删除过程。

13.6 思考与总结

- 主节点将快照命令放到集群状态中广播下去, 以此控制数据节点执行任务。数据节点执行完毕向主节点主动汇报状态。
- ES 的配置文件更新后不能动态生效。但是提供了 REST 接口来调整需要动态更新的参数。path.repo 字段需要写到配置文件中。当需要迁移数据时就要先改配置重启集群, 这样就不够方便。为什么不放在 REST 请求信息中, 而要求配置到文件里?
- 集群永久设置、模板都保存在集群状态中, 默认为不进行快照和恢复。注意索引别名不在集群状态中。快照默认会保存别名信息。
- Lucene 段合并会导致增量快照时产生新增内容。当段文件比较小时, 在 HDFS 中可能会产生许多小文件。因此通过 force_merge API 手工合并分段也有利于减少 HDFS 上的这些小文件。
- 快照写入了两个层面的元数据信息: 集群层和索引层。
- 快照与集群是否健康无关, 集群 Red 时也可以对部分索引执行快照。



- 数据复制过程中会计算校验和，确保复制后数据的正确性。
- 数据节点并发复制数据时取决于线程池的线程数的最大值，该值为 $\min(5, (\text{处理器数量}) / 2)$ 。
- 快照只对主分片执行。



14 chapter

第 14 章

Cluster 模块分析

Cluster 模块封装了在集群层面要执行的任务。例如，把分片分配给节点属于集群层面的工作，在节点间迁移分片以保持数据均衡，集群健康、集群级元信息管理，以及节点管理都属于集群层面工作。本章重点论述集群任务的执行，以及集群状态的下发过程。分片分配和节点管理等单独讨论更合适一些。

在 `_cluster/health` API 中看到的 `number_of_pending_tasks`（任务数量）就是等待执行的“集群任务”的任务数量，通过 `_cat/pending_tasks` API 可以列出具体的任务列表。本章介绍主节点都会执行哪些任务，以及任务的执行细节。这些任务由主节点执行，如果其他节点产生某些事件涉及集群层面的变更，则它需要向主节点发送一个 RPC 请求，然后由主节点执行集群任务。例如，在数据写入过程中，主分片写副分片失败，它会向主节点发送一个 RPC 请求，将副分片从同步分片列表中移除。

集群任务执行完毕，可能会产生新的集群状态。如果产生新的集群状态，则主节点会把它广播到其他节点。主节点和其他节点的通信使用最广泛的方式，就是通过下发集群状态让从节点执行相应的处理。控制信息、变更信息都存储在集群状态中。

我们先来看看集群状态中存在哪些内容。

14.1 集群状态

集群状态在 ES 中封装为 `ClusterState` 类。可以通过 `_cluster/state` API 来获取集群状态。

```
curl -X GET "localhost:9200/_cluster/state"
```



响应信息中提供了集群名称、集群状态的总压缩大小（下发到数据节点时是被压缩的）和集群状态本身，请求时可以通过设置过滤器来获取特定内容。

默认情况下，协调节点在收到这个请求后会吧请求路由到主节点执行，确保获取最新的集群状态。可以通过在请求中添加 `local=true` 参数，让接收请求的节点返回本地的集群状态。例如，在排查问题时如果怀疑节点的集群状态是否最新，则可以用这种方式来验证。

集群状态返回的信息比较多，为了节省篇幅，摘取关键信息如下。

```
{
  "cluster_name" : "elasticsearch",
  "compressed_size_in_bytes" : 1383, //压缩后的字节数
  "version" : 5, //当前集群状态的版本号
  "state_uuid" : "MMXpwaedThCVDIkzn9vpgA",
  "master_node" : "fc6s0S0hRi2yJvMo54qt_g",
  "blocks" : { }, //阻塞信息
  "nodes" : {
    "fc6s0S0hRi2yJvMo54qt_g" : {
      //节点名称、监听地址和端口等信息
    }
  },
  "metadata" : { //元数据
    "cluster_uuid" : "olrqNUxhTC2OVVG8KyXJ_w",
    "templates" : {
      //全部模板的具体内容
    },
    "indices" : { //索引列表
      "website" : {
        "state" : "open",
        "settings" : {
          //setting 的具体内容
        },
        "mappings" : {
          //mapping 的具体内容
        },
        "aliases" : [ ], //索引别名
        "primary_terms" : {
          //某个分片被选为主分片的次数，用于区分新旧主分片（具体请参考数据模型一章）
        }
      }
    }
  }
}
```



```
        "0" : 4,
        "1" : 5
    },
    "in_sync_allocations" : {
        //同步分片列表，代表某个分片中拥有最新数据的分片列表
        "1" : [
            "jalbFWjJST2bDPCUO08ScQ" //这个值是 allocation_id
        ],
        "0" : [
            "1EjTXElCSZ-C1DYL1EFRXtw"
        ]
    }
},
"repositories" : {
    //为存储快照而创建的仓库列表
},
"index-graveyard" : {
    //索引墓碑。记录已删除的索引，并保存一段时间。索引删除是主节点通过下发
    //集群状态来执行的
    //各节点处理集群状态是异步的过程。例如，索引分片分布在 5 个节点上，删除
    //索引期间，某个节点是“down”掉的，没有执行删除逻辑
    //当这个节点恢复的时候，其存储的已删除的索引会被当作孤立资源加入集群，
    //索引死而复活。墓碑的作用就是防止发生这种情况
    "tombstones" : [
        //已删除的索引列表
    ]
},
"routing_table" : { //内容路由表。存储某个分片位于哪个节点的信息。通过分片
    //找到节点
},
"indices" : { //全部索引列表
    "website" : { //索引名称
        "shards" : { //该索引的全部分片列表
            "1" : [ //分片 1
                {
                    "state" : "STARTED", //分片可能的状态: UNASSIGNED、INITIALIZING、
                    //STARTED、RELOCATING
                }
            ]
        }
    }
}
```




```
    "primary" : true, //是否是主分片
    "node" : "fc6s0S0hRi2yJvMo54qt_g", //所在节点
    "relocating_node" : null, //正在“relocating”到哪个节点
    "shard" : 1, //分片 1
    "index" : "website", //索引名
    "allocation_id" : {
        "id" : "jalbPWjJST2bDPCUO08ScQ" //分片唯一的 allocation_id 配合
                                           //in_sync_allocations 使用
    }
}
}
]
}
}
},
"routing_nodes" : { //存储某个节点存储了哪些分片的信息。通过节点找到分片
    "unassigned" : [ //未分配的分片列表
        { //某个分片的信息
            "state" : "UNASSIGNED",
            "primary" : true,
            "node" : null,
            "relocating_node" : null,
            "shard" : 0,
            "index" : "website",
            "recovery_source" : {
                "type" : "EXISTING_STORE"
            },
            "unassigned_info" : { //未分配的具体信息
                "reason" : "CLUSTER_RECOVERED",
                "at" : "2018-05-27T08:17:56.381Z",
                "delayed" : false,
                "allocation_status" : "no_valid_shard_copy"
            }
        }
    ],
    "nodes" : { //节点列表
        "fc6s0S0hRi2yJvMo54qt_g" : [ //某个节点上的分片列表
            {
```



```
        "state" : "STARTED", //分片信息, 同上
        "primary" : true,
        "node" : "fc6s0S0hRi2yJvMo54qt_g",
        "relocating_node" : null,
        "shard" : 1,
        "index" : "website",
        "allocation_id" : {
            "id" : "jalbPWjJST2bDPCU008ScQ"
        }
    }
}
]
}
},
"snapshot_deletions" : { //请求删除快照的信息
    "snapshot_deletions" : [ ]
},
"snapshots" : { //请求创建快照的信息
    "snapshots" : [ ]
},
"restore" : { //请求恢复快照的信息
    "snapshots" : [ ]
}
}
```

由于集群状态需要频繁下发, 而且内容较多, 从 ES 2.0 版本开始, 主节点发布集群信息时支持在相邻的两个版本号之间只发送增量内容。

14.2 内部封装和实现

下面介绍两个重要的类, `MasterService` 和 `ClusterApplierService` 分别负责运行任务和应用任务产生的集群状态。

14.2.1 MasterService

`MasterService` 类负责集群任务管理、运行等工作。其对外提供提交任务的接口, 内部维护一个线程池运行这些任务。对外提供的主要接口如下表所示。



方 法	简 介
numberOfPendingTasks	返回待执行的任务数量
pendingTasks	返回待执行的任务列表
submitStateUpdateTask	提交集群任务

主要数据成员如下表所示。

成 员	简 介
clusterStatePublisher	发布集群任务的模块
clusterStateSupplier	存储集群状态
slowTaskLoggingThreshold	集群任务慢执行的检测门限
threadPoolExecutor	执行集群任务的线程池
taskBatcher	管理、执行提交的任务。通过 submitStateUpdateTask 方法提交的任务最终调用内部类 Batcher 的提交方法

只有主节点会执行这个类中的方法。也就是说，只有主节点会提交集群任务到内部的队列，并运行队列中的任务。

14.2.2 ClusterApplierService

ClusterApplierService 类负责管理需要对集群任务进行处理的模块（Applier）和监听器（Listener），以及通知各个 Applier 应用集群状态。其对外提供接收集群状态的接口，当传输模块收到集群状态时，调用这个接口将集群状态传递过来，内部维护一个线程池用于应用集群状态。对外提供的主要接口如下表所示。

方 法	简 介
addStateApplier	添加一个对集群状态的处理器
addListener	添加一个集群状态监听器
removeApplier	删除一个集群状态处理器
removeListener	删除一个集群状态监听器
state	返回集群状态
onNewClusterState	收到新的集群状态
submitStateUpdateTask	在新的线程池应用集群状态

主要数据成员如下表所示。

成 员	简 介
clusterStateAppliers	保存要通知的集群状态应用处理器



续表

成 员	简 介
clusterStateListeners	保存集群状态监听器
state	保存最后的集群状态
threadPoolExecutor	应用集群任务的线程池

主节点和从节点都会应用集群状态，因此都会执行这个类中的方法。如果某个模块需要处理集群状态，则调用 `addStateApplier` 方法添加一个处理器。如果想监听集群状态的变化，则通过 `addListener` 添加一个监听器。`Applier` 负责将集群状态应用到组件内部，对 `Applier` 的调用在集群状态可见（`ClusterService#state` 获取到的）之前，对 `Listener` 的通知在新的集群状态可见之后。

实现一个 Applier

如果模块需要对集群状态进行处理，则需要从接口类 `ClusterStateApplier` 实现，实现其中的 `applyClusterState` 方法，例如：

```
public class GatewayMetaState extends AbstractComponent implements
ClusterStateApplier {
    public void applyClusterState(ClusterChangedEvent event) {
        //实现对集群状态的处理
    }
}
```

类似的，`SnapshotsService`、`RestoreService`、`IndicesClusterStateService` 都从 `ClusterStateApplier` 接口实现。

实现 `ClusterStateApplier` 的子类后，在子类中调用 `addStateApplier` 将类的实例添加到 `Applier` 列表。当应用集群状态时，会遍历这个列表通知各个模块执行应用。

```
clusterService.addStateApplier(this);
```

实现一个 Listener

大部分的组件只需要感知产生了新的集群状态，针对新的集群状态执行若干操作。如果模块需要在产生新的集群状态时被通知，则需要实现接口类 `ClusterStateListener`，实现其中的 `clusterChanged` 方法。例如：

```
public class GatewayService extends AbstractLifecycleComponent implements
ClusterStateListener {
```



```
public void clusterChanged(final ClusterChangedEvent event) {  
    //处理集群状态变化  
}  
}
```

类似的，实现 `ClusterStateListener` 接口的类还有 `IndicesStore`，`DanglingIndicesState`、`MetaDataUpdateSettingsService` 和 `SnapshotShardsService` 等。

实现 `ClusterStateListener` 的子类后，在子类中调用 `addListener` 将类的实例添加到 `Listener` 列表。当集群状态应用完毕，会遍历这个列表通知各个模块集群状态已发生变化。

```
clusterService.addListener(this);
```

14.2.3 线程池

运行集群任务、应用集群任务都在各自的线程池执行，它们的线程池的大小都是 1。

运行集群任务的线程池

执行集群任务的线程池为单个线程的线程池。因此，集群任务被串行地执行。线程池类型为 `PrioritizedEsThreadPoolExecutor`，支持优先级。其初始化在 `MasterService#doStart` 方法中：

```
threadPoolExecutor = EsExecutors.newSinglePrioritizing(MASTER_UPDATE_THREAD_NAME,  
    daemonThreadFactory(settings, MASTER_UPDATE_THREAD_NAME),  
threadPool.getThreadContext(), threadPool.scheduler());
```

该线程池的名称为 `masterService#updateTask`。

在 `newSinglePrioritizing` 方法中可以看到 `corePoolSize`、`maximumPoolSize` 都等于 1，`keepAliveTime` 为 0。

```
public static PrioritizedEsThreadPoolExecutor newSinglePrioritizing  
(String name, ThreadFactory threadFactory, ThreadContext contextHolder,  
ScheduledExecutorService timer) {  
    return new PrioritizedEsThreadPoolExecutor(name, 1, 1, 0L,  
TimeUnit.MILLISECONDS, threadFactory, contextHolder, timer);  
}
```

继续跟踪到 `PrioritizedEsThreadPoolExecutor` 构造函数，可以看到线程池使用带优先级的阻塞队列 `PriorityBlockingQueue`。



```
PrioritizedEsThreadPoolExecutor(String name, int corePoolSize, int
maximumPoolSize, long keepAliveTime, TimeUnit unit,
    ThreadFactory threadFactory, ThreadContext contextHolder,
ScheduledExecutorService timer) {
    super(name, corePoolSize, maximumPoolSize, keepAliveTime, unit, new
PriorityBlockingQueue<>(), threadFactory, contextHolder);
    this.timer = timer;
}
```

任务优先级定义在 `common.Priority` 中, 有 6 种类型的优先级: IMMEDIATE、URGENT、HIGH、NORMAL、LOW 和 LAZILY。默认为 NORMAL。

应用集群任务的线程池

应用集群任务的线程池也是单个线程的线程池, 集群任务被串行地应用。与运行集群任务时相同, 该线程池类型为 `PrioritizedEsThreadPoolExecutor`, 其初始化在 `ClusterApplierService#doStart` 方法中:

```
threadPoolExecutor = EsExecutors.newSinglePrioritizing(CLUSTER_UPDATE_
THREAD_NAME,
    daemonThreadFactory(settings, CLUSTER_UPDATE_THREAD_NAME),
threadPool.getThreadContext(), threadPool.scheduler());
```

该线程池的名称为 `clusterApplierService#updateTask`。

该线程池的构建过程及优先级信息与运行集群任务的线程池相同。

14.3 提交集群任务

什么情况下会提交集群任务? 通过查看对 `ClusterService#submitStateUpdateTask` 方法的调用, 可以看到哪些模块会提交集群任务, 这个任务会做什么事情也基本能看出来。下图并非完整调用。

GatewayService.java	236	clusterService.submitStateUpdateTask("local-gateway-elected-state", new ClusterStateUpdateTask() {
LocalAllocateDangledIndices.java	122	clusterService.submitStateUpdateTask("allocation dangled indices " + Arrays.toString(indexNames), new Cl
MasterDisruptionIT.java	217	internalCluster().getInstance(ClusterService.class, oldMasterNode).submitStateUpdateTask("sneaky-update
MetadataCreateIndexService.java	231	clusterService.submitStateUpdateTask("create-index [" + request.index() + "], cause [" + request.cause() +
MetadataDeleteIndexService.java	69	clusterService.submitStateUpdateTask("delete-index " + Arrays.toString(request.indices()),
MetadataIndexAliasesService.java	82	clusterService.submitStateUpdateTask("index-aliases",
MetadataIndexStateService.java	88	clusterService.submitStateUpdateTask("close-indices " + indicesAsString, new AckedClusterStateUpdateTa
MetadataIndexStateService.java	162	clusterService.submitStateUpdateTask("open-indices " + indicesAsString, new AckedClusterStateUpdateTa
MetadataIndexTemplateService.java	87	clusterService.submitStateUpdateTask("remove-index-template [" + request.name + "]", new ClusterStateU
MetadataIndexTemplateService.java	153	clusterService.submitStateUpdateTask("create-index-template [" + request.name + "], cause [" + request.ca



整理这些调用模块，直接或间接提交集群任务的时机可以归纳为以下几种情况。

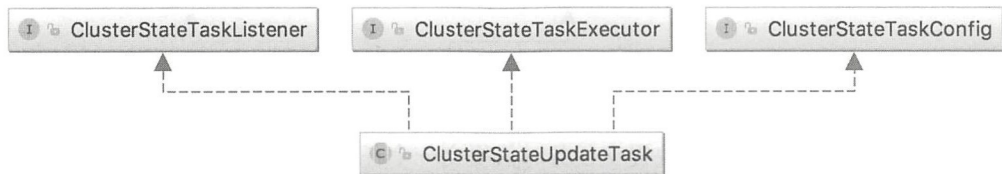
- 集群拓扑变化；
- 模板、索引 map、别名的变化；
- 索引操作：创建、删除、open、close；
- pipeline 的增删；
- 脚本的增删；
- gateway 模块发布选举出的集群状态；
- 分片分配；
- 快照、reroute api 等触发。

14.3.1 内部模块如何提交任务

内部模块通过 `clusterService.submitStateUpdateTask` 来提交一个集群任务。该方法有 3 种重载，参数大同小异，我们以最常见的方式为例来说明情况。函数原型如下：

```
public <T extends ClusterStateTaskConfig & ClusterStateTaskExecutor<T> &
ClusterStateTaskListener>
    void submitStateUpdateTask(String source, T updateTask)
```

第一个参数是事件源，第二个参数是要提交的具体的任务，这个任务有多种类封装，比如 `ClusterStateUpdateTask`、`AckedClusterStateUpdateTask`、`LocalClusterUpdateTask` 等，最具代表性是 `ClusterStateUpdateTask`，许多其他任务类型从它这里继承，或者与它的结构相似。`ClusterStateUpdateTask` 的类图结构如下图所示。



- **ClusterStateTaskListener**：提交任务时实现一些回调函数，例如，对任务处理失败、集群状态处理完毕时的处理。
- **ClusterStateTaskExecutor**：主要是定义要执行的任务。每个任务在执行时会传入当前集群状态，任务执行完毕返回新产生的集群状态。如果没有产生新的集群状态，则返回原集群状态的实例。





- **ClusterStateTaskConfig**: 任务的配置信息，包括超时和优先级。

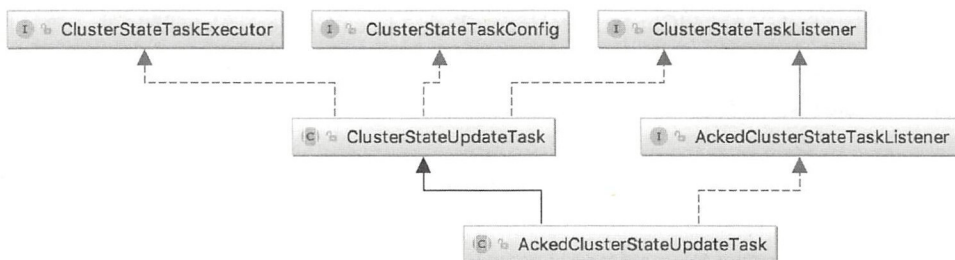
任务类 `ClusterStateUpdateTask` 实现了这三个接口，总体来说就是要定义执行的任务，以及实现一些事件通知函数。当某个模块提交任务时，使用下面的简单方式：

```
clusterService.submitStateUpdateTask("allocation dangled indices ", new
ClusterStateUpdateTask() {
    //实现要执行的具体任务，任务返回新的集群状态
    public ClusterState execute(ClusterState currentState) {
    }

    //任务执行失败的回调
    public void onFailure(String source, Exception e) {
    }

    //集群状态处理完毕的回调，当集群状态已经被全部 Appliers 和 Listeners 处理完毕后调用
    public void clusterStateProcessed(String source, ClusterState oldState,
ClusterState newState) {
    }
});
```

任务类 `AckedClusterStateUpdateTask` 在 `ClusterStateUpdateTask` 的基础上增加了更多的回调通知，其类结构如下图所示。



在绝大部分情况下，各个模块提交任务时只提交一个任务，只有两种情况下可能会提交多个任务，都在选主流程中：

1. ElectionContext#closeAndBecomeMaster

```
Map<DiscoveryNode, ClusterStateTaskListener> tasks = getPendingAsTasks();
tasks.put(BECOME_MASTER_TASK, (source1, e) -> {}); // noop listener, the
election finished listener determines result
```





```
tasks.put(FINISH_ELECTION_TASK, electionFinishedListener);
masterService.submitStateUpdateTasks(source, tasks, ClusterStateTaskConfig.
build(Priority.URGENT), joinTaskExecutor);
```

2. ElectionContext#closeAndProcessPending

```
Map<DiscoveryNode, ClusterStateTaskListener> tasks = getPendingAsTasks();
tasks.put(FINISH_ELECTION_TASK, electionFinishedListener);
masterService.submitStateUpdateTasks(source, tasks, ClusterStateTaskConfig.
build(Priority.URGENT), joinTaskExecutor);
```

两处的 `getPendingAsTasks` 返回的并非执行集群任务线程池的等待队列，而是选主流程中等待加入集群的请求数量。

现在我们来看一下 `submitStateUpdateTask` 具体执行过程。

14.3.2 任务提交过程实现

无论提交单个任务，还是提交多个任务，`submitStateUpdateTask` 最终通过 `TaskBatcher#submitTasks` 来提交任务。该方法的实现有些难以理解。其核心工作就是将提交的任务交给线程池执行。但是在此基础上实现了部分场景下的任务去重。

去重的原理是：拥有相同 `ClusterStateTaskExecutor` 对象实例的任务只执行一个。

去重在实现时并没有在线程池队列的列表上操作，而是将任务列表添加到一个独立于线程池任务队列之外的 `HashMap` 中：`tasksPerBatchingKey`。这个 `HashMap` 保存的也是待执行的任务。与线程池任务队列不同的是，当提交多个任务时，线程池队列中只保存第一个，而 `tasksPerBatchingKey` 以 `ClusterStateTaskExecutor` 为 Key，Value 是提交的整个任务列表。当从线程池任务队列中取出任务准备执行时，先根据任务的 `ClusterStateTaskExecutor` 从 `tasksPerBatchingKey` 中找到它的任务列表，然后批量执行这个任务列表。所谓批量执行实际上只执行一个，然后对其他任务赋予相同的执行结果。

换句话说，区分重复任务的方式是通过定义的任务本身实现的，即定义具有相同的 `ClusterStateTaskExecutor` 对象。去重的方式并非将重复的任务从列表中删除，而是在执行完任务后赋予重复任务相同的执行结果。

去重的时机有两方面：

- (1) 提交的任务列表本身的去重。
- (2) 提交的任务在任务队列 `tasksPerBatchingKey` 中已存在，也就是存在尚未执行的相同任





务，此时新提交的任务被迫加到 `tasksPerBatchingKey` 某个 `k` 对应的 `v` 中。

提交的任务列表本身任务数量大于 1 个的情况如上一小节所述，只在选主的流程中。那么，拥有相同 `ClusterStateTaskExecutor` 对象的是什么情况？一般情况下，各模块调用 `clusterService.submitStateUpdateTask` 提交任务，如果在任务执行之前提交多次，则不认为这些是重复的任务。因为一般每次提交会创建不同的 `ClusterStateUpdateTask` 对象，因此也拥有不同的 `ClusterStateTaskExecutor` 实例。例如，提交两次相同的创建索引请求，两次请求都会被加入任务队列，然后执行 2 次。

只有模块在提交任务时明确使用了相同的 `ClusterStateTaskExecutor` 对象，才可能会在执行时去重。例如，每次提交 `shard-started` 任务时使用创建好的 `ClusterStateTaskExecutor`，因此是同一个 `ClusterStateTaskExecutor` 对象实例。

```
clusterService.submitStateUpdateTask(  
    "shard-started " + request,  
    request,  
    ClusterStateTaskConfig.build(Priority.URGENT),  
    shardStartedClusterStateTaskExecutor,  
    shardStartedClusterStateTaskExecutor);  
channel.sendResponse(TransportResponse.Empty.INSTANCE);  
}
```

下面我们看一下任务提交过程 `submitTasks` 的具体实现。

`submitTasks` 的核心工作就是将待执行任务加入任务队列。每个任务都有优先级，线程池的任务队列也是支持优先级的 `PriorityBlockingQueue`。

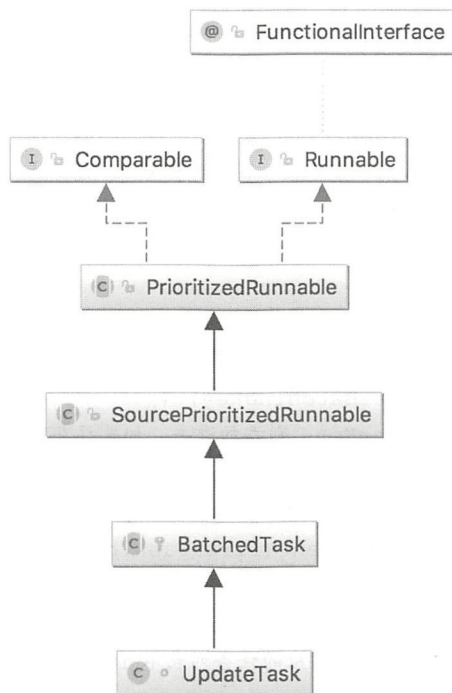
各个入口的 `submitStateUpdateTask` 最终通过 `TaskBatcher` 提交任务：

```
taskBatcher.submitTasks(safeTasks, config.timeout());
```

第一个参数为任务列表，第二个参数为超时时间。提交的任务本质上是一个 `Runnable`。

`UpdateTask` 的类图结构如下图所示。





submitTasks 方法的主要实现过程如下：

```

public void submitTasks(List<? extends BatchedTask> tasks, @Nullable
TimeValue timeout) throws EsRejectedExecutionException {
    //batchingKey 为 ClusterStateTaskExecutor
    //BatchedTask::getTask 返回完整的 task <T extends ClusterStateTaskConfig
    //& ClusterStateTaskExecutor<T> & ClusterStateTaskListener>
    //例如, ClusterStateUpdateTask 对象

    final BatchedTask firstTask = tasks.get(0);

    //如果一次提交多个任务, 则必须有相同的 batchingKey, 这些任务将被批量执行
    assert tasks.stream().allMatch(t -> t.batchingKey == firstTask.
batchingKey) :
        "tasks submitted in a batch should share the same batching key: " +
tasks;

    //将 tasks 从 List 转换为 Map, key 为 task 对象, 例如, ClusterStateUpdateTask,
    //如果提交的任务列表存在重复则抛出异常

```

```

    final Map<Object, BatchedTask> tasksIdentity = tasks.stream().collect
(Collectors.toMap(
    BatchedTask::getTask,
    Function.identity(),
    (a, b) -> { throw new IllegalStateException("cannot add
duplicate task: " + a); },
    IdentityHashMap::new));

//tasksPerBatchingKey 在这里添加, 在任务执行线程池中, 在任务真正开始运行之前 "remove"
//key 为 batchingKey, 值为 tasks
synchronized (tasksPerBatchingKey) {
    //如果不存在 batchingKey, 则添加进去, 如果存在则不操作
    //computeIfAbsent 返回新添加的 k 对应的 v, 或者已存在的 k 对应的 v
    LinkedHashSet<BatchedTask> existingTasks =
tasksPerBatchingKey.computeIfAbsent(firstTask.batchingKey,
    k -> new LinkedHashSet<>(tasks.size()));
    for (BatchedTask existing : existingTasks) {
        //一个 batchingKey 对应的任务列表不可有相同的 identity, identity 是任务
        //本身, 例如, ClusterStateUpdateTask 对象
        BatchedTask duplicateTask = tasksIdentity.get(existing.getTask());
        if (duplicateTask != null) {
            throw new IllegalStateException();
        }
    }
    //添加到 tasksPerBatchingKey 的 value 中。如果提交了相同的任务,
    //则新任务被追加到这里
    existingTasks.addAll(tasks);
}

//交给线程执行
if (timeout != null) {
    threadExecutor.execute(firstTask, timeout, () -> onTimeoutInternal
(tasks, timeout));
} else {
    threadExecutor.execute(firstTask);
}
}

```



虽然只将传入任务列表的第一个任务交给线程池执行，但是任务列表的全部任务被添加到 `tasksPerBatchingKey` 中，线程池执行任务时，根据任务的 `batchingKey` 从 `tasksPerBatchingKey` 中获取任务列表，然后批量执行这个任务列表。

当调用 `threadExecutor.execute` 将其交任务给线程池执行时，如果线程池中没有任何任务运行，则立刻执行这个任务，否则将任务加入线程池的任务队列。

14.4 集群任务的执行过程

`masterService#updateTask` 线程池负责执行集群任务，任务的执行入口为 `TaskBatcher.BatchedTask#run`，但是线程池队列中的这个任务并非真实要执行的任务，而是任务的一部分。

1. 构建任务列表

先根据任务的 `batchingKey` 从提交任务时记录的 `HashMap` (`tasksPerBatchingKey`) 中提取要执行的任务列表。这个任务列表是拥有相同 `ClusterStateTaskExecutor` 的任务。用这个任务列表中尚未执行的任务构建新的列表，这个新列表就是真实要执行的任务列表：

```
void runIfNotProcessed(BatchedTask updateTask) {
    if (updateTask.processed.get() == false) {
        synchronized (tasksPerBatchingKey) {
            //根据 batchingKey 获取任务列表
            LinkedHashSet<BatchedTask> pending = tasksPerBatchingKey.remove(
                updateTask.batchingKey);
            if (pending != null) {
                for (BatchedTask task : pending) {
                    if (task.processed.getAndSet(true) == false) {
                        //构建要执行的任务列表
                        toExecute.add(task);
                        processTasksBySource.computeIfAbsent(task.source, s ->
                            new ArrayList<>()).add(task);
                    }
                }
            }
        }

        if (toExecute.isEmpty() == false) {
            //执行任务列表
        }
    }
}
```



```

        run(updateTask.batchingKey, toExecute, tasksSummary);
    }
}
}

```

这个任务列表接下来被批量执行，由于要执行的内容相同，所以列表中的任务只会执行一个。

2. 执行任务

执行任务并发布集群状态的总体过程在 `MasterService#runTasks` 方法中实现。在该方法中，在执行任务前先确认本节点是否为主节点，然后将当前的集群状态传递给任务执行函数，在 `MasterService#executeTasks` 中回调提交任务时定义 `execute` 方法，任务执行完毕，将任务列表赋予相同的执行结果。

```

public final ClusterTasksResult<...> throws Exception {
    //执行提交任务时定义 execute 方法
    ClusterState result = execute(currentState);
    //将任务列表赋予相同的执行结果
    return ClusterTasksResult.<ClusterStateUpdateTask>builder().successes
(tasks).build(result);
}

```

如果任务执行失败，则回调 `onFailure` 方法通知 `Listener`。如果执行成功并产生新的集群状态，则进入集群状态发布过程。

3. 发布集群状态

通过 `clusterStatePublisher` 发布集群状态，并在成功或失败后通知相关回调函数：

```

protected void runTasks(TaskInputs taskInputs) {
    //是否产生新的集群状态
    if (taskOutputs.clusterStateUnchanged()) {
    }
    else{
        try {
            //发布集群状态
            clusterStatePublisher.accept(clusterChangedEvent, taskOutputs.
createAckListener(threadPool, newClusterState));
        } catch (Discovery.FailedToCommitClusterStateException t) {

```



```

        //回调 listener.onFailure
        taskOutputs.publishingFailed(t);
        return;
    }
    //集群状态发布成功, 回调 listener.clusterStateProcessed
    taskOutputs.processedDifferentClusterState(previousClusterState,
newClusterState);
    //回调 clusterStatePublished
    taskOutputs.clusterStatePublished(clusterChangedEvent);
}
}

```

14.5 集群状态的发布过程

发布集群状态是一个分布式事务操作, 分布式事务需要实现原子性: 要么所有参与者都提交事务, 要么都取消事务。ES 使用二段提交来实现分布式事务。二段提交可以避免失败回滚, 其基本过程是: 把信息发下去, 但不应用, 如果得到多数节点的确认, 则再发一个请求出去要求节点应用。

ES 实现二段提交与标准二段提交有一些区别, 发布集群状态到参与者的数量并非定义为全部, 而是多数节点成功就算成功。多数的定义取决于配置项:

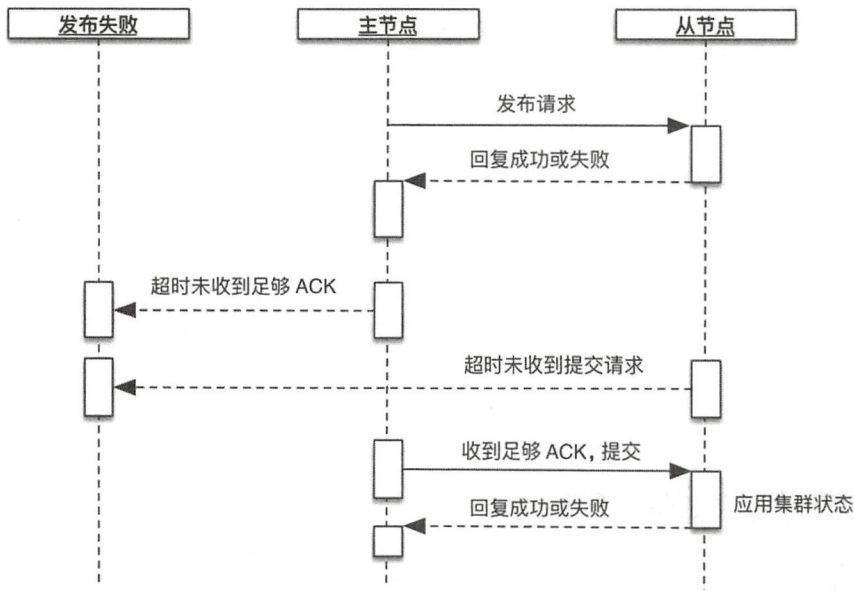
```
discovery.zen.minimum_master_nodes
```

两个阶段过程如下。

- 发布阶段: 发布集群状态, 等待响应。
- 提交阶段: 收到的响应数量大于 `minimum_master_nodes` 数量, 发送 `commit` 请求。

二段提交图例如下图所示。





二段提交不能保证第二阶段节点收到 `commit` 请求后正确应用事务。它只能保证参与者都提交了事务，不能保证事务在单个节点上提交成功还是失败，因此事务的执行可能在部分节点失败。

主节点把集群状态发下去，节点收到后不应用，当主节点在 `discovery.zen.commit_timeout` 超时时间内，收到来节点的确认数量达到 `discovery.zen.minimum_master_nodes-1`（减去本节点）时，主节点开始发送提交请求，如果节点在 `discovery.zen.commit_timeout` 超时时间内没有收到主节点提交请求，则拒绝该集群状态。

当节点收到主节点的提交请求后，开始应用集群状态。主节点等待响应，直到收到全部响应（注意是全部，而非 `discovery.zen.minimum_master_nodes-1`），或者达到 `discovery.zen.publish_timeout` 超时，整个发布流程结束。`discovery.zen.publish_timeout` 计时器是从发送第一个请求时就开始计时的。

与集群状态发布相关的配置项如下表所示。

配 置	简 介
<code>discovery.zen.publish_timeout</code>	整个集群状态的发布超时时间，默认为 30 秒
<code>discovery.zen.commit_timeout</code>	第一阶段之后，等待节点确认的超时时间，默认为 30 秒
<code>discovery.zen.publish_diff.enable</code>	是否允许以增量方式发布集群状态，默认为 <code>true</code>
<code>discovery.zen.no_master_block</code>	当集群无主时，该选项决定哪些操作被拒绝。有两个值可以设置： <code>all</code> 为所有操作，读写都被拒绝； <code>write</code> 为写操作被拒绝、读可以成功。基于最后的已知集群状态，可能会读到陈旧的信息。默认设置为 <code>write</code>



发布集群状态的实现入口在 `ZenDiscovery#publish`，我们重点关注发布过程中的几个问题，增量发布是如何实现的？二段提交是如何实现的？

14.5.1 增量发布的实现原理

每个集群状态都有自己唯一的版本号，ES 在发布集群状态时允许在相邻版本号之间只发送增量内容。在发布之前的准备工作中，先准备发布集群状态的目的节点列表，这个列表只是在已知集群列表中移除了本地节点。

```
nodes = clusterChangedEvent.state().nodes();
for (final DiscoveryNode node : nodes) {
    if (node.equals(localNode) == false) {
        nodesToPublishTo.add(node);
    }
}
```

遍历这个节点列表，如果上次发布时这个节点成功了，则给它准备增量的信息，否则准备全量的信息。

```
private void buildDiffAndSerializeStates(...) {
    Diff<ClusterState> diff = null;
    for (final DiscoveryNode node : nodesToPublishTo) {

        if (sendFullVersion || !previousState.nodes().nodeExists(node)) {
            //准备全量内容
            if (serializedStates.containsKey(node.getVersion()) == false) {
                serializedStates.put(node.getVersion(), serializeFullClusterState(
                    clusterState, node.getVersion()));
            }
        } else {
            //准备增量内容
            if (diff == null) {
                diff = clusterState.diff(previousState);
            }
            if (serializedDiffs.containsKey(node.getVersion()) == false) {
                serializedDiffs.put(node.getVersion(),
                    serializeDiffClusterState(diff, node.getVersion()));
            }
        }
    }
}
```



```

        }
    }

}

```

增量和全量的集群状态都会被序列化并压缩，全量的信息保存在 `serializedStates` 中，增量的信息保存在 `serializedDiffs` 中。

此时，对于发布过程来说的必备信息：目标节点列表，以及增量或全量数据已准备完毕。进入发布过程：`PublishClusterStateAction#innerPublish`。

14.5.2 二段提交总流程

二段提交发送的第一个请求是发布的集群状态数据。在发送第一个请求的响应处理中检查是否达到发送第二个请求的条件，触发提交阶段。

```

private void innerPublish(...) {
    //publish_timeout 计时器开始
    final long publishingStartInNanos = System.nanoTime();

    //遍历节点，异步发送全量或增量内容，这是二段提交的第一个请求
    for (final DiscoveryNode node : nodesToPublishTo) {
        if (sendFullVersion || !previousState.nodes().nodeExists(node)) {
            sendFullClusterState(clusterState, serializedStates, node,
publishTimeout, sendingController);
        } else {
            sendClusterStateDiff(clusterState, serializedDiffs, serializedStates,
node, publishTimeout, sendingController);
        }
    }
}

//等待第一个请求收到的响应足够，或者达到 commit_timeout 超时时间，如果超
//时后仍未收到足够数量的响应，则抛出异常，结束发布过程
sendingController.waitForCommit(discoverySettings.getCommitTimeout());

//第一阶段已正常完成，等待第二个请求，也就是提交请求完成
try {

```

```

        long timeLeftInNanos = Math.max(0, publishTimeout.nanos() -
(System.nanoTime() - publishingStartInNanos));
        final BlockingClusterStatePublishResponseHandler publishResponseHandler
= sendingController.getPublishResponseHandler();
        //等待提交请求收到足够的回复, 或者达到 publish_timeout 超时
        sendingController.setPublishingTimedOut(!publishResponseHandler.
awaitAllNodes(TimeValue.timeValueNanos(timeLeftInNanos)));
    } catch (InterruptedException e) {
        // ignore & restore interrupt
        Thread.currentThread().interrupt();
    }
}

```

14.5.3 发布过程

无论发送全量还是增量内容, 最终都通过 `PublishClusterStateAction#sendClusterStateToNode` 实现发送。

该方法调用 `transportService` 异步发送数据, 并在请求的响应中判断是否可以进入提交阶段, 整体过程如下:

```

private void sendClusterStateToNode(...) {
    try {
        transportService.sendRequest(... new EmptyTransportResponseHandler() {
            public void handleResponse(TransportResponse.Empty response) {
                //检查收到响应是否足够, 符合条件是发送提交请求
                sendingController.onNodeSendAck(node);
            }
            public void handleException(TransportException exp) {
                sendingController.onNodeSendFailed(node, exp);
            }
        });
    } catch (Exception e) {
        sendingController.onNodeSendFailed(node, e);
    }
}

```

`handleException` 的处理只是将计数器 `latch` 减一, 该计数器的总大小为发布过程要通知的节

点总数: `nodesToPublishTo`。该计数器用于判断整个发布过程是否已结束: `publishTimeout` 计时器超时, 或者该计数器为 0。

在发布请求的响应处理中, 当收到足够数量的 (`discovery.zen.minimum_master_nodes - 1`) 节点正常返回的响应数量 (即 `handleResponse` 的数量, 如果主节点在发布阶段收到节点异常返回则不计算在其中), 主节点发送提交请求。如果太多节点返回异常, 则导致没有收到足够数量的 (`discovery.zen.minimum_master_nodes - 1`) 节点确认, 不会执行提交过程。

下面验证是否满足条件, 执行提交的过程:

```
private synchronized void checkForCommitOrFailIfNoPending(DiscoveryNode
masterNode) {
    neededMastersToCommit--;
    //收到足够的正常的响应数量后发送提交请求
    if (neededMastersToCommit == 0) {
        if (markAsCommitted()) {
            for (DiscoveryNode nodeToCommit : sendAackedBeforeCommit) {
                //发送提交请求
                sendCommitToNode(nodeToCommit, clusterState, this);
            }
            sendAackedBeforeCommit.clear();
        }
    }
    decrementPendingMasterAcksAndChangeForFailure();
}
```

14.5.4 提交过程

当条件满足后, 正式发送提交请求, 发送过程仍然通过 `transportService` 异步发送。

```
private void sendCommitToNode(...) {
    try {
        transportService.sendRequest(...
            new EmptyTransportResponseHandler() {
                //记录收到的响应数量, 收到足够数量的响应, 或 publish_timeout 超时
                //后, 发布过程结束
                public void handleResponse(TransportResponse.Empty response) {
                    sendingController.getPublishResponseHandler()
                        .onResponse(node);
                }
            }
        );
    }
```



```

    }
    public void handleException(TransportException exp) {
        sendingController.getPublishResponseHandler().onFailure
(node, exp);
    }
    });
} catch (Exception t) {
    sendingController.getPublishResponseHandler().onFailure(node, t);
}
}
}

```

在对请求的响应处理上, `handleResponse` 与 `handleException` 本质上执行相同的处理, `onResponse` 和 `onFailure` 只是将计数器 `latch` 减一, 该计数器与上一小节的相同。如果某个节点应用失败了, 则只能让它失败, 其他节点正常应用集群状态。

14.5.5 异常处理

什么算发布成功, 什么算发布失败? 除了准备数据阶段将集群状态序列化, 压缩过程产生的 I/O 异常等内部错误, 成功与否取决于二段提交的执行结果。二段提交过程只有一次中止分布式事务的机会, 就是在提交阶段, 没有收到足够节点 ACK (回复正常的响应)。在这种情况下, 主节点会重新加入集群。

一旦进入提交阶段, 发布过程就进入不可逆状态, 如果有节点应用失败了, 则整个发布过程不会被认为失败。即使只有少数节点正常应用集群状态, 最终也只能接受这种结果。

如果从节点在应用集群状态时中止, 例如, 节点被 “kill”, 服务器断电等异常, 则节点的集群状态应用失败。当节点重启之后, 主动从 Master 节点获取最新的集群状态并应用。

14.6 应用集群状态

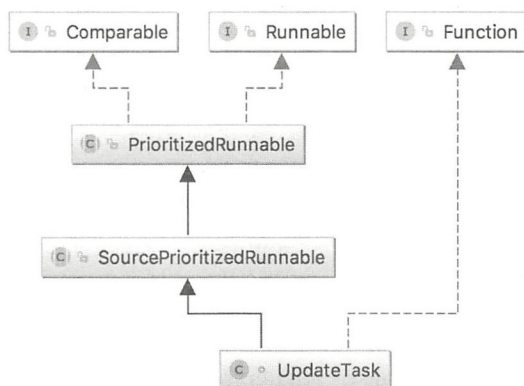
新的集群状态生成之后, 主节点和其他从节点都会应用集群状态。主节点在发布集群状态时在目的节点列表中排除了自己, 但是在发布成功之后会应用这个集群状态。主节点应用集群状态在 `ZenDiscovery#publish` 方法中, 发布成功之后执行, 从节点在对集群状态的响应处理在 `PublishClusterStateAction#handleCommitRequest` 方法中, 主从节点最终都调用 `ZenDiscovery#processNextCommittedClusterState` 应用集群状态。

应用集群状态, 通知要对集群状态处理的模块, 封装在 `ClusterApplierService` 类中。当准备应用一个集群状态, 在做完一些异常处理后, 调用该类的 `onNewClusterState` 方法进入应用过程,

并在 `submitStateUpdateTask` 方法中将集群状态应用任务交给线程池执行：

```
private void submitStateUpdateTask(...) {
    UpdateTask updateTask = new UpdateTask(config.priority(), source, new
    SafeClusterStateTaskListener(listener, logger), executor);
    threadPoolExecutor.execute(updateTask);
}
```

`UpdateTask` 类本质上是一个线程任务（`Runnable`），其类结构如下图所示。



线程池 `clusterApplierService#updateTask` 运行任务时执行 `ClusterApplierService#runTask` 方法，同样先进行异常检查，例如，检查应用的集群状态是否与上一个相同，如果相同则不应用。在正常情况下，进入 `applyChanges` 应用集群状态，该方法中正式调用各个模块的 `Applier` 和 `Listener`，并更新本节点存储的集群状态。

```
private void applyChanges(UpdateTask task, ClusterState previousClusterState,
    ClusterState newClusterState) {
    //通知所有 Appliers
    callClusterStateAppliers(clusterChangedEvent);
    //更新本类存储的集群状态信息
    state.set(newClusterState);
    //通知所有 Listeners
    callClusterStateListeners(clusterChangedEvent);
    //回调 clusterStateProcessed
    task.listener.clusterStateProcessed(task.source, previousClusterState,
    newClusterState);
}
```

在调用各模块对集群状态进行处理时，具体方式就是遍历 `Applier`，依次调用各模块的处理函数，没有先后顺序关系：

```
private void callClusterStateAppliers(ClusterChangedEvent clusterChangedEvent) {
    //遍历全部的 Applier，依次调用各模块对集群状态的处理
    clusterStateAppliers.forEach(applier -> {
        try {
            //调用各模块实现的 applyClusterState
            applier.applyClusterState(clusterChangedEvent);
        } catch (Exception ex) {
            //某个模块应用集群状态出现异常时打印日志，但应用过程不会终止
            logger.warn("failed to notify ClusterStateApplier", ex);
        }
    });
}
```

同理，当执行完各模块的 `applyClusterState` 后，遍历通知所有的 `Listener`：

```
private void callClusterStateListeners(ClusterChangedEvent clusterChangedEvent) {
    //遍历通知全部 Listener
    Stream.concat(clusterStateListeners.stream(),
        timeoutClusterStateListeners.stream()).forEach(listener -> {
        try {
            //调用各模块实现的 clusterChanged
            listener.clusterChanged(clusterChangedEvent);
        } catch (Exception ex) {
            //某个模块执行出现异常时打印日志，但通知过程不会终止
            logger.warn("failed to notify ClusterStateListener", ex);
        }
    });
}
```

14.7 查看等待执行的集群任务

通过 `/_cluster/health` API 可以查看等待任务的总数，该 API 用于简单地查看一些汇总信息。等待任务数为返回信息的 `number_of_pending_tasks` 字段。其返回信息如下：

```
{
    "cluster_name" : "testcluster",
```

```

    "status" : "yellow",
    "timed_out" : false,
    "number_of_nodes" : 1,
    "number_of_data_nodes" : 1,
    "active_primary_shards" : 5,
    "active_shards" : 5,
    "relocating_shards" : 0,
    "initializing_shards" : 0,
    "unassigned_shards" : 5,
    "delayed_unassigned_shards": 0,
    "number_of_pending_tasks" : 0,
    "number_of_in_flight_fetch": 0,
    "task_max_waiting_in_queue_millis": 0,
    "active_shards_percent_as_number": 50.0
  }

```

更进一步地，如果想查看等待执行的具体是什么任务，则可以使用 `/_cluster/pending_tasks` 或 `/_cat/pending_tasks` API，它们返回的信息基本相同。例如：

```

{
  "tasks": [
    {
      "insert_order": 101,
      "priority": "URGENT",
      "source": "create-index [foo_9], cause [api]",
      "time_in_queue_millis": 86,
      "time_in_queue": "86ms"
    }
  ]
}

```

返回信息中包含了任务插入顺序、优先级，以及具体的任务名称。`source` 字段是提交任务时第一个参数指定的，表示提交任务的源，通过它可以看出要执行什么样的任务。

14.8 任务管理 API

与集群层面执行的任务不同，任务管理 API 中的“任务”不一定是集群层面的任务，一般

指周期性执行的任务，或者用户发起的任务。例如，node stats、search queries、create index 等，如果一个用户发起的任务引起了集群状态的变化，则这个任务可能会同时被 task API 和上一节的 pending cluster tasks API 列出。

14.8.1 列出运行中的任务

任务管理 API 允许检索当前在集群中的一个或多个节点上执行的任务的信息。下面的命令可以返回集群所有节点正在运行的任务：

```
curl -X GET "localhost:9200/_tasks"
```

也可以查看特定节点上运行的任务，下面的命令返回在节点 nodeId1 和 nodeId2 上运行的所有任务：

```
curl -X GET "localhost:9200/_tasks?nodes=nodeId1,nodeId2"
```

还可以过滤出与集群相关的任务，下面的命令返回节点 nodeId1 和 nodeId2 上运行的所有与集群相关的任务：

```
curl -X GET "localhost:9200/_tasks?nodes=nodeId1,nodeId2&actions=cluster:*"
```

还可以使用 actions=*search&detailed 返回正在执行的搜索任务，detailed 参数表示返回详细信息。返回信息大致如下：

```
{
  "nodes" : {
    "oTUltX4IQMOUUVeiohTt8A" : {
      "name" : "H5dfFeA",
      "transport_address" : "127.0.0.1:9300",
      "host" : "127.0.0.1",
      "ip" : "127.0.0.1:9300",
      "tasks" : {
        "oTUltX4IQMOUUVeiohTt8A:464" : {
          "node" : "oTUltX4IQMOUUVeiohTt8A",
          "id" : 464,
          "type" : "transport",
          "action" : "indices:data/read/search",
          "description" : "indices[test], types[test], search_type[QUERY_THEN_FETCH], source[{\"query\":...}]",
```



```
        "start_time_in_millis" : 1483478610008,  
        "running_time_in_nanos" : 13991383,  
        "cancellable" : true  
    }  
}  
}  
}  
}
```

14.8.2 取消任务

如果任务支持被取消，则可以使用下面的方式取消一个长时间运行的任务：

```
curl -X POST "localhost:9200/_tasks/node_id:task_id/_cancel"
```

任务取消命令支持与列出任务相同的任务选择参数，因此可以同时取消多个任务。例如，下面的命令将取消在节点 `nodeId1` 和 `nodeId2` 上运行的所有重建索引任务。

```
curl -X POST "localhost:9200/_tasks/_cancel?nodes=nodeId1,nodeId2&actions=  
*reindex"
```

本节只是给出了任务管理 API 的基本使用方式，完整命令请参考官方手册：<https://www.elastic.co/guide/en/elasticsearch/reference/master/tasks.html>。

14.9 思考与总结

(1) 对任务的去重只在提交任务的调用方明确定义了相同 `ClusterStateTaskExecutor` 对象的情况下有效。

(2) 二段提交的成功意味着 `minimum_master_nodes` 个数的节点执行了对集群状态的应用，但无法保证节点是否正常应用，如果应用时产生错误，则不会认为发布失败。

15 chapter

第 15 章

Transport 模块分析

传输模块用于集群内节点之间的内部通信。从一个节点到另一个节点的每个调用都使用传输模块。例如，当一个节点处理 HTTP GET 请求时，实际上是由持有该数据的另一个节点处理的，这就需要处理 HTTP GET 请求的节点将请求通过传输模块转发给另一个节点。

传输机制是完全异步的，这意味着没有阻塞线程等待响应。使用异步通信的好处是解决了 C10k 问题，也是广播请求/收集结果（例如，ES 中的搜索）的理想解决方案。

15.1 配置信息

15.1.1 传输模块配置

TCP transport 是传输模块基于 TCP 的实现，有以下配置，如下表所示。

配 置	简 介
transport.tcp.port	绑定端口范围。默认为 9300~9400
transport.publish_port	集群中其他节点在与此节点通信时应使用的端口。当集群节点位于代理或防火墙后面，并且在 transport.tcp.port 不能从外部直接访问时，此功能很有用。默认为通过 transport.tcp.port 分配的实际端口
transport.bind_host	传输服务绑定的主机地址。默认为 transport.host（如果明确设置）或 network.bind_host

续表

配 置	简 介
transport.publish_host	节点在集群上发布的主机地址，其他节点连接这个地址。默认为 transport.host（如果明确设置）或 network.publish_host
transport.host	用于设置 transport.bind_host 和 transport.publish_host。默认设置为 transport.host 或 network.host
transport.tcp.connect_timeout	套接字连接超时设置。默认为 30 秒
transport.tcp.compress	设置为 true，以在所有节点之间启用压缩（LZF）。默认为 false
transport.ping_schedule	定期发送 ping 消息，以确保连接是活跃的。传输客户端默认为 5 秒，设置成 -1 为禁用

默认情况下，传输模块使用 9300 端口通信。该端口承载了三种不同的业务：客户端的 Java API 通信，节点间的内部通信，以及自动发现的通信。使用 transport profiles，可以将三者配置到不同的地址和端口。例如：

```
transport.profiles.default.port: 9300-9400
transport.profiles.default.bind_host: 10.0.0.1
transport.profiles.client.port: 9500-9600
transport.profiles.client.bind_host: 192.168.0.1
transport.profiles.dmz.port: 9700-9800
transport.profiles.dmz.bind_host: 172.16.1.2
```

这在部分场景下很有用，例如，想保护集群内部通信的 9300 端口，只对外开放 9200 端口，但又不想影响 Java API 的使用，那么可以为 JavaAPI 单独配置一个服务端口。

传输模块有一个专用的 tracer 日志，当它被激活时，日志系统会记录传入和传出的请求。可以通过动态设置 org.elasticsearch.transport.TransportService.tracer 为 TRACE 级别来开启：

```
curl -X PUT "localhost:9200/_cluster/settings" -H 'Content-Type:
application/json' -d'
{
  "transient" : {
    "logger.org.elasticsearch.transport.TransportService.tracer" : "TRACE"
  }
}
```

还可以使用一组 include 和 exclude 通配符模式来控制 tracer 的内容。默认情况下，除了 ping

的故障检测请求，所有请求都将被跟踪：

```
curl -X PUT "localhost:9200/_cluster/settings" -H 'Content-Type: application/
json' -d'
{
  "transient" : {
    "transport.tracer.include" : "*",
    "transport.tracer.exclude" : "internal:discovery/zen/fd*"
  }
}
```

15.1.2 通用网络配置

除了传输模块配置，还有一些其他的网络配置信息。

```
network.host
```

节点将绑定到此主机名或 IP 地址，并将此主机公布到集群中的其他节点。接受一个 IP 地址、主机名、一个特殊值，或者是这些内容任意组合的数组。默认为 `_local_`。

```
discovery.zen.ping.unicast.hosts
```

为了加入集群，节点需要知道集群中一些其他节点的主机名或 IP 地址。此设置提供节点将尝试联系其他节点的初始列表，接受 IP 地址或主机名。如果主机名被解析为多个 IP 地址，那么每个 IP 地址都将用于 discovery。轮询 DNS 也可以在 discovery 中使用，每次返回一个不同的 IP 地址，如果 IP 地址不存在则抛出异常，并在下一轮 ping 时重新解析（取决于 JVM DNS 缓存）。默认值为 `["127.0.0.1", "::1"]`。

```
http.port
```

用于接受 HTTP 请求的绑定端口。接受单个值或范围。如果指定了一个范围，则该节点将绑定到该范围内的第一个可用端口。默认为 9200~9300。

```
transport.tcp.port
```

为节点之间的通信绑定端口。接受单个值或范围。如果指定了一个范围，则该节点将绑定到该范围内的第一个可用端口。默认为 9300~9400。

`network.host` 允许设置以下特殊值，如下表所示。

特 殊 值	简 介
<code>_[networkInterface]_</code>	网络接口地址，例如， <code>_en0_</code>
<code>_local_</code>	代表回环地址：127.0.0.1
<code>_site_</code>	系统上的任意站点—本地地址，例如，192.168.0.1
<code>_global_</code>	系统上任意全局范围的地址，例如，8.8.8.8

默认情况下，这些特殊的值在 IPv4 和 IPv6 上都可以正常工作，但是也可以使用 `ipv4` 或 `ipv6` 说明符来明确指定。例如，`_en0:ipv4_` 只绑定到 `en0` 的 IPv4 地址。

`network.host` 是常用的设置监听地址的方式，同时设置绑定主机和发布主机。在一些高级用例中，可能需要为它们设置不同值。

```
network.bind_host
```

指定节点应该绑定到哪个网络接口，以便监听传入的请求。一个节点可以绑定到多个接口，例如，两个网卡，或者一个站点本地地址和一个回环地址。默认为 `network.host`。

```
network.publish_host
```

发布主机是节点向集群中发布的单个网口，以便其他节点可以连接它。目前，ES 可以绑定到多个地址，但只发布一个地址。如果没有指定，则默认为来自 `network.host` 的“最佳”地址。按 IPv4/IPv6 栈优先排序，然后是可达性。如果为 `network.host` 设置多个地址，但在节点到节点的通信中依赖于特定的地址，那么应该显式地设置 `network.publish_host`。

以上两个配置都可以像 `network.host` 一样配置。它们接收 IP 地址、主机名和特殊值。

基于 TCP 的所有组件（如 HTTP 和传输模块）共享以下高级设置，如下表所示。

配 置	简 介
<code>network.tcp.no_delay</code>	<code>no_delay</code> 设置，默认为 <code>true</code>
<code>network.tcp.keep_alive</code>	<code>keep_alive</code> 设置，默认为 <code>true</code>
<code>network.tcp.reuse_address</code>	<code>reuse</code> 选项，在非 Windows 系统上默认为 <code>true</code>
<code>network.tcp.send_buffer_size</code>	发送缓冲区大小，单位为字节，默认没有设置
<code>network.tcp.receive_buffer_size</code>	接收缓冲区大小，单位为字节，默认没有设置

15.2 Transport 总体架构

ES 的传输模块和 HTTP 传输模块都是基于 Netty 实现的。Netty 是一个 Java 实现的高性能异步网络通信库，基于 `epoll/kqueue` 实现事件处理。

我们说的传输模块，目前只有一种实现，就是 TCP 传输模块。如上节所述，TCP 传输模块有三类用处：内部节点间通信（我们称为 RPC）、Java API 客户端，以及节点发现。HTTP 模块负责服务用户的 REST 请求。

15.2.1 网络层

网络层是对内部各种传输模块的抽象，使得上层发送/接收数据时不必关心底层的实现，使用 Netty 还是其他类库，上层并不关心。在内部实现中，传输模块和 HTTP 模块统一封装到 `NetworkModule` 类中。顾名思义，该类是在 TCP 传输模块和 HTTP 传输模块之上封装的，实现了对各种传输模块的初始化，上层的发送和接收依赖对网络模块的引用。

该类的几个重要数据成员如下表所示。

成 员	简 介
<code>Transport</code>	负责内部节点的 RPC 请求等
<code>HttpServerTransport</code>	负责对用户的 REST 接口服务
<code>TransportInterceptor</code>	传输层拦截器，允许插件在发送和接收数据时拦截请求（目前只实现了发送拦截）

上述三个成员在 `NetworkModule` 的构造函数（节点启动时调用）中通过插件方式加载。

主要对外接口如下表所示。

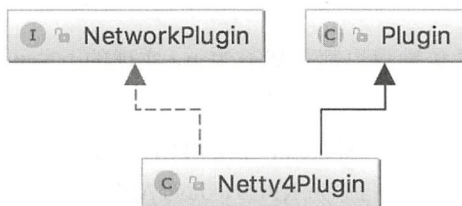
方 法	简 介
<code>getTransportSupplier</code>	返回注册的传输模块
<code>getHttpServerTransportSupplier</code>	返回注册的 HTTP 传输模块
<code>getTransportInterceptor</code>	返回注册的所有拦截器

1. 网络模块初始化

初始化 `NetworkModule` 传输模块和 HTTP 传输模块之后，上层就可以通过该类对外提供的接口获取某个传输模块了。该类返回的各种传输模块和拦截器都是虚类型，因此本质上就是对网络层的一个抽象。

`NetworkModule` 内部组件的初始化是通过插件方式加载的。在其构造函数中传入 `NetworkPlugin`

列表, NetworkPlugin 是一个接口类, Netty4Plugin 从这个接口实现, 如下图所示。



在 Netty4Plugin 中, 分别构建了 Netty4Transport 和 Netty4HttpServerTransport, 用于传输模块和 HTTP 传输模块:

```

public class Netty4Plugin extends Plugin implements NetworkPlugin {
    //构建 Netty4Transport 作为 Transport
    public Map<String, Supplier<Transport>> getTransports(...) {
        return Collections.singletonMap(NETTY_TRANSPORT_NAME, () -> new
Netty4Transport(settings, threadPool, networkService, bigArrays,
        namedWriteableRegistry, circuitBreakerService));
    }
    //构建 Netty4HttpServerTransport 作为 HttpServerTransport
    public Map<String, Supplier<HttpServerTransport>> getHttpTransports(...) {
        return Collections.singletonMap(NETTY_HTTP_TRANSPORT_NAME,
            () -> new Netty4HttpServerTransport(settings, networkService,
bigArrays, threadPool, xContentRegistry, dispatcher));
    }
}

```

根据加载的 NetworkPlugin 插件和定义好的 REST 处理器初始化 NetworkModule:

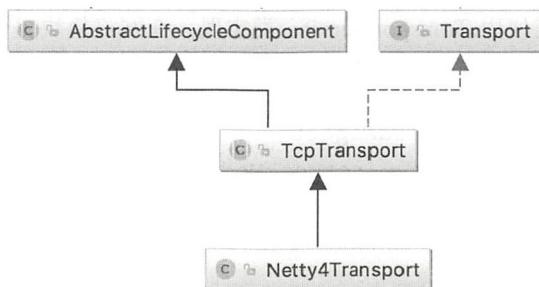
```

//已注册好的 REST 请求处理器
final RestController restController = actionModule.getRestController();
//初始化 NetworkModule
final NetworkModule networkModule = new NetworkModule(settings, false,
pluginsService.filterPlugins(NetworkPlugin.class),
    threadPool, bigArrays, circuitBreakerService, namedWriteableRegistry,
xContentRegistry, networkService, restController);

```

2. Netty4Transport

Netty4Transport 用于 RPC 等内部通信, 其继承自 TcpTransport, 类结构如下图所示。



在初始化时构建了 Client 和 Server:

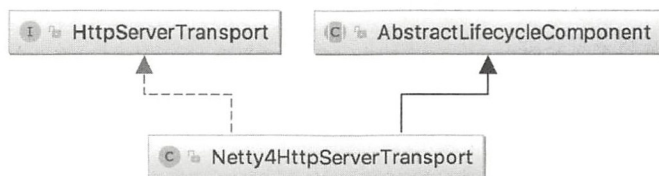
```

protected void doStart() {
    //初始化 Client
    bootstrap = createBootstrap();
    if (NetworkService.NETWORK_SERVER.get(settings)) {
        for (ProfileSettings profileSettings : profileSettings) {
            //初始化 Server
            createServerBootstrap(profileSettings);
            bindServer(profileSettings);
        }
    }
}

```

3. Netty4HttpServerTransport

Netty4HttpServerTransport 用于响应 REST 请求，其继承自 HttpServerTransport，如下图所示。



同样在 Netty4HttpServerTransport#doStart 中创建一个 HTTP Server 监听端口，当收到用户请求时，调用 dispatchRequest 对不同的请求执行相应的处理。哪种请求被哪个类处理这种信息注册在 ActionModule 类中。

15.2.2 服务层

服务层指网络模块的上层应用层，基于网络模块提供的 Transport 来收/发数据。本节重点

分析节点间通信，该通信使用 `TransportService` 类实现，在网络模块提供的 `Transport` 基础上，该类提供连接到节点、发送数据、注册事件响应函数等方法。其初始化过程如下：

```
//通过网络模块获取已初始化的 Transport
final Transport transport = networkModule.getTransportSupplier().get();
//基于网络模块的 Transport 构建 TransportService
final TransportService transportService = newTransportService(settings,
    transport, threadPool,
        networkModule.getTransportInterceptor(), localNodeFactory,
    settingsModule.getClusterSettings());
```

在节点内部任何通信前，首先需要连接到集群的其他节点。

1. 连接到节点

在默认配置下，ES 的每个节点与其他节点都会保持 13 个长连接。每个连接有各自的用途。可以通过配置调节某种业务使用的连接数。

当本节点连接到某个特定的节点时，`TransportService` 通过网络层提供的 `transport.connectToNode` 完成连接。在完成连接后，内部维护一个 `NodeChannels` 类对象，表示节点到节点的连接。其中包括多个 TCP 连接（默认为 13 个），并记录了每个连接的用途。目前，这些连接有以下几类用途，定义在 `TransportRequestOptions.Type` 中。

```
public enum Type {
    RECOVERY, //用于恢复
    BULK, //用于批量写入
    REG, //其他用途，例如，加入集群等
    STATE, //传输集群状态
    PING //用作 nodeFD 或 masterFD 的 ping 请求
}
```

这些连接被 `ConnectionProfile` 统一管理：

```
static ConnectionProfile buildDefaultConnectionProfile(Settings settings) {
    //默认为 2 个
    int connectionsPerNodeRecovery = CONNECTIONS_PER_NODE_RECOVERY.get(
        settings);
    //默认为 3 个
    int connectionsPerNodeBulk = CONNECTIONS_PER_NODE_BULK.get(settings);
    //默认为 6 个
```

```

    int connectionsPerNodeReg = CONNECTIONS_PER_NODE_REG.get(settings);
    //默认为 1 个
    int connectionsPerNodeState = CONNECTIONS_PER_NODE_STATE.get(settings);
    //默认为 1 个
    int connectionsPerNodePing = CONNECTIONS_PER_NODE_PING.get(settings);
    ConnectionProfile.Builder builder = new ConnectionProfile.Builder();
    builder.addConnections(connectionsPerNodeBulk,
TransportRequestOptions.Type.BULK);
    //...
    return builder.build();
}

```

节点间每种用途的连接数量可以通过以下配置进行调整：

```

transport.connections_per_node.recovery
transport.connections_per_node.bulk
transport.connections_per_node.reg
transport.connections_per_node.state
transport.connections_per_node.ping

```

NodeChannels 类保存了这些连接，当发送数据时，根据请求类型找到对应的连接来发送数据。

```

public final class NodeChannels implements Connection {
    //保存每个请求对应的连接
    private final Map<TransportRequestOptions.Type, ConnectionProfile.
ConnectionTypeHandle> typeMapping;
    //保存已建立的 TCP 连接
    private final List<TcpChannel> channels;
    //目的节点是哪个
    private final DiscoveryNode node;
}

```

建立连接过程如下，如果 13 个连接中有一个连接失败，则整体认为失败，关闭已建立的连接。

```

public final NodeChannels openConnection(DiscoveryNode node, ConnectionProfile
connectionProfile) throws IOException {
    //获取总连接数
    int numConnections = connectionProfile.getNumConnections();
}

```



```
List<TcpChannel> channels = new ArrayList<>(numConnections);
for (int i = 0; i < numConnections; ++i) {
    try {
        //建立一个连接
        TcpChannel channel = initiateChannel(node, connectionProfile.
getConnectTimeout(), connectFuture);
        channels.add(channel);
    } catch (Exception e) {
        //如果产生异常, 则关闭所有连接
        TcpChannel.closeChannels(channels, false);
        throw e;
    }
}
//构建并返回 NodeChannels
nodeChannels = new NodeChannels(node, channels, connectionProfile,
version);
return nodeChannels;
}
```

2. 发送请求

内部的 RPC 请求通过 `TransportService#sendRequest` 发送, 在之前的章节中经常可以见到这种发送请求的调用。

`sendRequest` 会检查目的节点是否是本节点, 如果是本节点, 则在 `sendLocalRequest` 方法中直接通过 `action` 获取对应的 `Handler`, 调用对应的处理函数。简化的实现过程如下:

```
private void sendLocalRequest(...) {
    final RequestHandlerRegistry reg = getRequestHandler(action);
    reg.processMessageReceived(request, channel);
}
```

当需要发送到网络时, 调用 `asyncSender.sendRequest` 执行发送, 最终通过 `TcpTransport.NodeChannels#sendRequest` 发送数据, 先根据请求类型获取相应的连接, 某种类型如果有多个连接, 例如, BULK 请求, 则会在多个连接中轮询选择。

```
public void sendRequest(long requestId, String action, TransportRequest
request, TransportRequestOptions options){
    //通过请求类型获取 13 个连接中的相应连接
```

```
TcpChannel channel = channel(options.type());
//发送请求
sendRequestToChannel(this.node, channel, requestId, action, request,
options, getVersion(), (byte) 0);
}
```

3. 定义对 Response 的处理

当通过 `TransportService#sendRequest` 发送一个 RPC 请求时, 本节点作为 RPC 客户端, 需要同时定义当服务器执行完 RPC, 返回 `Response` 后, 对这个 `Response` 如何处理。`TransportResponseHandler` 类负责定义对这个 `Response` 的处理。在发送请求的 `sendRequest` 函数的最后一个参数中定义, 例如:

```
transportService.sendRequest(primaryNode, IN_FLIGHT_OPS_ACTION_NAME, new
InFlightOpsRequest(shardId),
    new TransportResponseHandler<InFlightOpsResponse>() {
        //返回一个新的 response
        public InFlightOpsResponse newInstance() {
            return new InFlightOpsResponse();
        }

        //对远程节点执行成功的处理
        public void handleResponse(InFlightOpsResponse response) {
            listener.onResponse(response);
        }

        //对远程节点执行失败的处理
        public void handleException(TransportException exp) {
            logger.debug("{} unexpected error while retrieving in flight
op count", shardId);
            listener.onFailure(exp);
        }

        //返回线程池名称
        public String executor() {
            return ThreadPool.Names.SAME;
        }
    });
```

TransportResponseHandler 类主要定义了对远程节点执行 RPC 请求后返回成功还是失败处理。

4. 定义对请求的处理

本节点也需要处理来自其他节点的 RPC 请求，因此需要定义对每个 RPC 使用哪个模块进行处理。具体参考 RPC 一节。

15.3 REST 解析和处理

对 REST 请求的处理就是定义某个 URI 应该由哪个模块处理。在 ES 中，REST 请求和 RPC 请求都称为 Action，对 REST 请求执行处理的类命名规则为 Rest*Action。

ActionModule 类中注册了某个类对某个 REST 请求的处理，并对外提供这种映射关系，每个 REST 处理类继承自 RestNodesInfoAction。处理类在 ActionModule 中的注册过程如下：

```
public void initRestHandlers(Supplier<DiscoveryNodes> nodesInCluster) {
    registerHandler.accept(new RestMainAction(settings, restController));
    registerHandler.accept(new RestNodesInfoAction(settings, restController,
settingsFilter));
    registerHandler.accept(new RestRemoteClusterInfoAction(settings,
restController));
    //省略大部分的注册
}
```

以 RestNodesInfoAction 为例，在构造函数中注册对某个 URI 的处理如下。

```
public RestNodesInfoAction(Settings settings, RestController controller,
SettingsFilter settingsFilter) {
    super(settings);
    controller.registerHandler(GET, "/_nodes", this);
    controller.registerHandler(GET, "/_nodes/{nodeId}", this);
    this.settingsFilter = settingsFilter;
}
```

同时，每个 REST 请求处理类需要实现一个 prepareRequest 函数，用于在收到请求时，对请求执行验证工作等，当一个请求到来时，网络层调用 BaseRestHandler#handleRequest。在这个函数中，会调用子类的 prepareRequest，然后执行这个 Action：

```
public final void handleRequest(RestRequest request, RestChannel channel,
NodeClient client) throws Exception {
    ///调用子类的 prepareRequest
    final RestChannelConsumer action = prepareRequest(request, client);
    //执行子类定义的任务
    action.accept(channel);
}
```

对 Action 的具体处理定义在处理类的 prepareRequest 的 Lambda 表达式中, 例如:

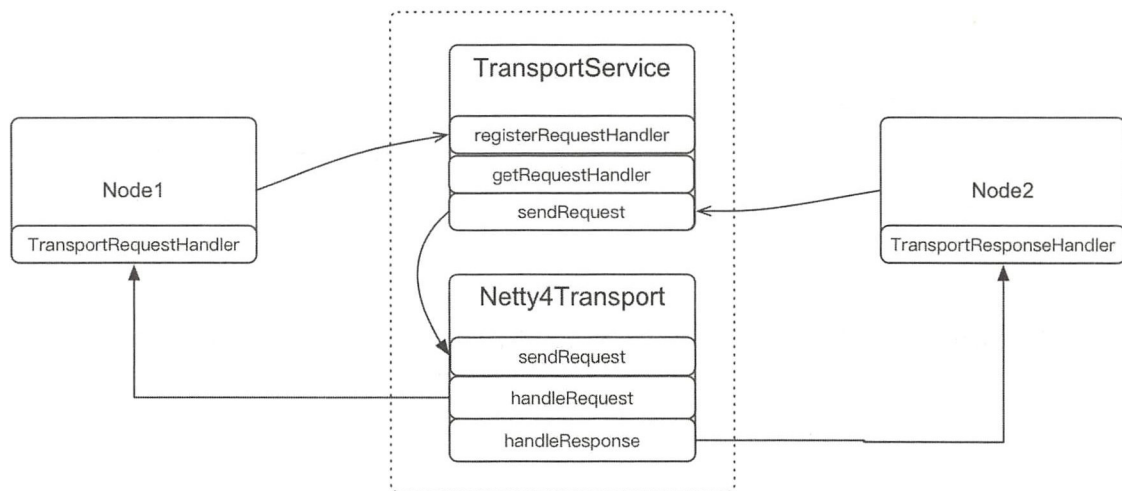
```
public RestChannelConsumer prepareRequest(final RestRequest request, final
NodeClient client) throws IOException {
    //省略对请求的预处理
    //...
    //Action 具体要执行的任务
    return channel -> client.admin().cluster().getSnapshots
(getSnapshotsRequest, new RestToXContentListener<>(channel));
}
```

当 Netty 收到 HTTP 请求后, 调用 Netty4HttpServerTransport#dispatchRequest, 该方法根据定义好的 Action 调用对应的 Rest*Action 处理类。

15.4 RPC 实现

RPC 是远程过程调用的简称, 当一个节点需要另一个节点执行某些操作时, 例如, 创建、删除索引等, 向这个节点发送一个 RPC 请求, ES 的 RPC 基于 TCP 实现, 底层是 Netty 的 Netty4Transport。每个 RPC 在内部称为 Action, 有唯一的名称, 例如, cluster:monitor/main。当传输模块收到一个 RPC 请求时, 会根据这个 Action 名称获取对应的处理类。

TransportService 类是在网络层之上对 RPC 的发送与接收的服务层封装, 虽然从模块设计角度来说, 网络层的设计对内部是否使用 Netty 框架是解耦的, 除 Netty 外, 也可以使用其他通信框架, 但是为了让读者更容易理解, 我们看一下从 TransportService 到 Netty4Transport 的联系, 如下图所示。



在上图中, Node2 调用 `sendRequest` 发送请求, 发送时传入定义好的 `TransportResponseHandler`, `TransportService` 调用 `Netty4Transport` 的 `sendRequest` 发送数据。当远程节点处理完毕, `Netty4Transport` 的 `handleResponse` 方法最终回调发送请求时定义的 `TransportResponseHandler`。

Node1 接收请求, 通过 `registerRequestHandler` 注册 `Action` 和对应的处理类 `TransportRequestHandler`。`TransportService` 类中维护了 `Action` 与处理类的对应关系。当 `Netty4Transport` 收到请求后, `handleRequest` 方法中调用 `TransportService` 类的 `getRequestHandler(action)` 通过 (客户端请求中的) `Action` 获取相应的处理类, 最终调用 `TransportRequestHandler` 执行对 RPC 的处理逻辑。

15.4.1 RPC 的注册和映射

一个 RPC 请求与处理模块的对应关系在两方面维护:

- 在 `ActionModule` 类中注册 `Action` 与处理类的映射;
- 通过 `TransportService#registerRequestHandler` 方法注册 `Action` 名称与对应处理器的映射。

这两种映射关系同时存在。RPC 先在 `ActionModule` 类中注册, 再调用 `TransportService#registerRequestHandler` 在 `TransportService` 类中注册。在大部分情况下, 网络层收到请求后根据 `TransportService` 注册的 `Action` 信息获取对应的处理模块。

1. ActionModule 类中的注册

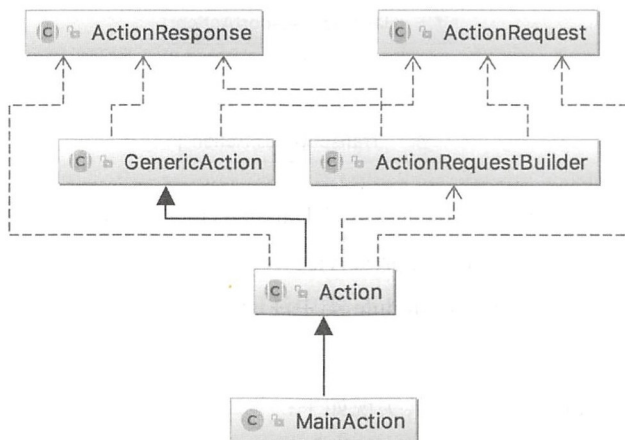
与 REST `Action` 的注册类似, 内部 RPC 也注册在 `ActionModule` 类中, 描述某个 `Action` 应该被哪个类处理。一个 `Action` 可以理解为 RPC 调用的名称。`Action` 与处理类的映射关系如下:


```

static Map<String, ActionHandler<?, ?>> setupActions(List<ActionPlugin>
actionPlugins) {
    ActionRegistry actions = new ActionRegistry();
    actions.register(MainAction.INSTANCE, TransportMainAction.class);
    actions.register(NodesInfoAction.INSTANCE,
TransportNodesInfoAction.class);
    actions.register(RemoteInfoAction.INSTANCE,
TransportRemoteInfoAction.class);
    //省略大部分 Action 的注册
}

```

register 函数的第一个参数是名称规则为*Action 的类，以比较简单的 MainAction 为例，其类结构如下图所示。



这个类主要定义了 Action 的名称及返回响应，同样以 MainAction 为例，其实现如下：

```

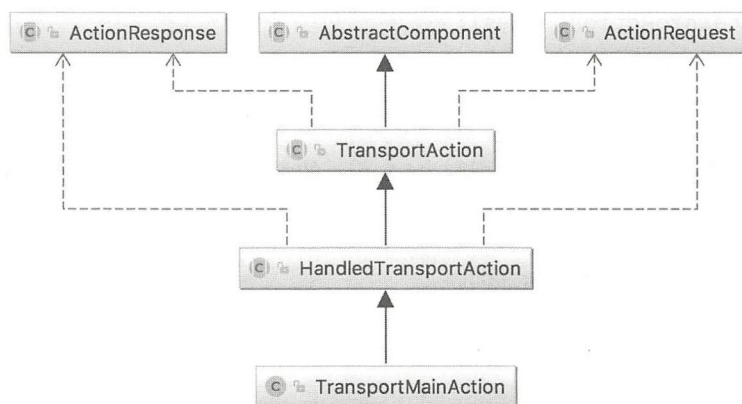
public class MainAction extends Action<MainRequest, MainResponse,
MainRequestBuilder> {
    //定义 Action 名称，后续会根据 Action 名称找到对应的处理类
    public static final String NAME = "cluster:monitor/main";
    public static final MainAction INSTANCE = new MainAction();
    public MainRequestBuilder newRequestBuilder(ElasticsearchClient client) {
        return new MainRequestBuilder(client, INSTANCE);
    }

    public MainResponse newResponse() {

```

```
        return new MainResponse();  
    }  
}
```

第二个参数是名称规则为 `Transport*Action` 的类, 在这个类中定义对此 `Action` 的具体处理。以 `TransportMainAction` 类为例, 其类结构如下图所示。



注意, 许多 `Transport*Action` 类都会继承自 `HandledTransportAction`。而在 `HandledTransportAction` 类的构造函数中, 会调用 `TransportService#registerRequestHandler` 在 `TransportService` 类中注册处理器。因此, 许多在 `ActionModule` 类中注册的 RPC 信息会自动在 `TransportService` 中添加映射关系。

以 `TransportMainAction` 类为例, 其实现如下:

```
public class TransportMainAction extends HandledTransportAction  
<MainRequest, MainResponse> {  
    public TransportMainAction() {  
    }  
  
    //定义对此 RPC 的处理  
    protected void doExecute(MainRequest request, ActionListener<MainResponse>  
listener) {  
    }  
}
```

`doExecute` 函数汇总需要实现最重要的对 RPC 请求的具体处理逻辑。

2. TransportService 类中的注册

在 TransportService 中注册 RPC 信息是为了在收到传输层的请求后，通过 Action 字符串找到对应的处理类。注册过程需要提供两个关键信息：Action 名称与请求处理类（TransportRequestHandler 对象）。

在 TransportService 类中通过 registerRequestHandler 注册 RPC 信息的来源可以分为两种，一种是来自 ActionModule 的注册，Transport*Action 类的父类 HandledTransportAction 在其构造函数中自动调用 registerRequestHandler；另一种是来自其他模块的直接调用 registerRequestHandler，例如，TransportReplicationAction 和 MasterFaultDetection。

以 HandledTransportAction 类在 TransportService 中的注册为例，其注册过程如下：

```
transportService.registerRequestHandler(actionName, request,
ThreadPool.Names.SAME, false, canTripCircuitBreaker,
    new TransportHandler());
```

TransportService 将映射维护在一个 Map 中：

```
Map<String, RequestHandlerRegistry>
```

Map 的 key 为 Action 名称，RequestHandlerRegistry 中封装了与 RPC 相关的 Action 名称、处理器等信息。通过它可以找到对应的处理模块。为 registerRequestHandler 传入的最后一个参数就是定义的处理器。这个处理器需要从 TransportRequestHandler 类继承。如前所述，在 TransportService 类中注册 RPC 的时机来源于 ActionModule 和 HandledTransportAction 的构造函数，我们以 HandledTransportAction 构造函数中注册时为例，其处理器定义如下：

```
class TransportHandler implements TransportRequestHandler<Request> {
    //当收到 RPC 请求
    public final void messageReceived(final Request request, final
TransportChannel channel, Task task) throws Exception {
        //执行这个 RPC 请求
        execute(task, request, new ActionListener<Response>() {
            //执行成功，将产生的响应回复客户端
            public void onResponse(Response response) {
                channel.sendResponse(response);
            }

            //执行失败，向客户端回复失败原因
            public void onFailure(Exception e) {
```

```

        channel.sendResponse(e);
    }
    });
}
}

```

对请求的处理方法 `execute` 定义在 `TransportAction` 类中，它先检测请求的合法性，然后调用 `Transport*Action` 中定义的 `doExecute` 函数执行真正的 RPC 处理逻辑。

```

public final void execute(Task task, Request request, ActionListener
<Response> listener) {
    //验证请求
    ActionRequestValidationException validationException = request.validate();
    RequestFilterChain<Request, Response> requestFilterChain = new
RequestFilterChain<>(this, logger);
    //调用 Action 定义的 doExecute 函数执行用户定义的处理
    requestFilterChain.proceed(task, actionName, request, listener);
}

```

15.4.2 根据 Action 获取处理类

当收到一个 RPC 请求进行处理时，由于触发点的不同，有多种途径找到这个 RPC 对应的处理模块是哪个。

1. REST 请求触发

某些 REST 请求会触发内部的 RPC 请求，在这种情况下，在 `NodeClient#executeLocally` 方法中通过 Action 获取 `TransportAction`，Actions 是 `ActionModule` 类中注册的 RPC 列表。

```

TransportAction transportAction(GenericAction<Request, Response> action) {
    TransportAction<Request, Response> transportAction = actions.get
(action);
    return transportAction;
}

```

获取 `TransportAction` 后，调用 `execute` 执行处理。处理过程与上一节所述相同，在 `requestFilterChain.proceed` 方法中调用此 Action 的 `doExecute` 函数进行处理。

2. TcpTransport 收到 RPC 请求

从 TCP 传输层（也就是 9300 端口）收到一个 RPC 请求是最常见的方式。当收到一个请求时，首先在 `TcpTransport#messageReceived` 中进行基本的处理，然后到 `handleRequest` 方法中处理请求，在这个方法中，调用 `TransportService` 通过 `Action` 获取处理类。

```
//通过 Action 获取处理模块
final RequestHandlerRegistry reg = transportService.getRequestHandler(action);
//调用处理模块执行对 RPC 的处理逻辑
threadPool.executor(reg.getExecutor()).execute(new RequestHandler(reg,
request, transportChannel));
```

15.5 思考与总结

（1）本章主要分析了 REST API 和内部 RPC 的解析与调用，以及网络层与服务层的关系。

（2）默认情况下，ES 的每个节点与其他节点都保持 13 个长连接，这在集群规模较大时，例如，达到 1000 节点时，会维护非常多的连接。在这种情况下，如果重新启动集群，由于需要在短时间内建立大量连接，则新建连接的请求有可能被操作系统认为是 SYN 攻击。

16 chapter

第 16 章

ThreadPool 模块分析

每个节点都会创建一系列的线程池来执行任务，许多线程池都有与其相关任务队列，用来允许挂起请求，而不是丢弃它。下面列出目前 ES 版本中的线程池。

`generic`

用于通用的操作（例如，节点发现），线程池类型为 `scaling`。

`index`

用于 `index/delete` 操作，线程池类型为 `fixed`，大小为处理器的数量，队列大小为 200，允许设置的最大线程数为 1+处理器数量。

`search`

用于 `count/search/suggest` 操作。线程池类型为 `fixed`，大小为 $\text{int}((\text{处理器数量} \times 3) / 2) + 1$ ，队列大小为 1000。

`get`

用于 `get` 操作。线程池类型为 `fixed`，大小为处理器的数量，队列大小为 1000。

`bulk`

用于 `bulk` 操作，线程池类型为 `fixed`，大小为处理器的数量，队列大小为 200，该线程池允许设置的最大线程数为 1+处理器数量。

`snapshot`

用于 `snapshot/restore` 操作。线程池类型为 `scaling`，线程保持存活时间为 5min，最大线程数

为 $\min(5, (\text{处理器数量})/2)$ 。

warmer

用于 segment warm-up 操作。线程池类型为 scaling，线程保持存活时间为 5min，最大线程数为 $\min(5, (\text{处理器数量})/2)$ 。

refresh

用于 refresh 操作。线程池类型为 scaling，线程空闲保持存活时间为 5min，最大线程数为 $\min(10, (\text{处理器数量})/2)$ 。

listener

主要用于 Java 客户端线程监听器被设置为 true 时执行动作。线程池类型为 scaling，最大线程数为 $\min(10, (\text{处理器数量})/2)$ 。

same

在调用者线程执行，不转移到新的线程池。

management

管理工作的线程池，例如，Node info、Node tats、List tasks 等。

flush

用于索引数据的 flush 操作。

force_merge

顾名思义，用于 Lucene 分段的 force merge。

fetch_shard_started

用于 TransportNodesAction。

fetch_shard_store

用于 TransportNodesListShardStoreMetaData。

线程池和队列的大小可以通过配置文件进行调整，例如，为 search 增加线程数和队列大小：

```
thread_pool.search.size: 30
```

16.1 线程池类型

如同任何要并发处理任务的服务程序一样，线程池要处理的任务类型大致可以分为两类：CPU 计算密集型和 I/O 密集型。对于两种不同的任务类型，需要为线程池设置不同的线程数量。

一般说来，线程池的大小可以参考如下设置，其中 N 为 CPU 的个数：

- 对于 CPU 密集型任务，线程池大小设置为 $N+1$ ；
- 对于 I/O 密集型任务，线程池大小设置为 $2N+1$ 。

对于计算密集型任务，线程池的线程数量一般不应该超过 $N+1$ 。如果线程数量太多，则会导致更高的线程间上下文切换的代价。加 1 是为了当计算线程出现偶尔的故障，或者偶尔的 I/O、发送数据、写日志等情况时，这个额外的线程可以保证 CPU 时钟周期不被浪费。

I/O 密集型任务的线程数可以稍大一些，因为 I/O 密集型任务大部分时间阻塞在 I/O 过程，适当增加线程数可以增加并发处理能力。而上下文切换的代价相对来说已经不那么敏感。但是线程数量不一定设置为 $2N+1$ ，具体要看 I/O 等待时间有多长。等待时间越长，需要越多的线程，等待时间越少，需要越少的线程。因此也可以参考下面的公式：

$$\text{最佳线程数} = ((\text{线程等待时间} + \text{线程 CPU 时间}) / \text{线程 CPU 时间}) \times \text{CPU 数}$$

为了应对这两种类型的任务，ES 封装了以下类型的线程池。

16.1.1 fixed

线程池拥有固定数量的线程来处理请求，当线程空闲时不会销毁，当所有线程都繁忙时，请求被添加到队列中。

size 参数用来控制线程的数量。

queue_size 参数用来控制线程池相关的任务队列大小。设置为 -1 表示无限制。当请求到达时，如果队列已满，则请求将被拒绝。

例如：

```
thread_pool.search.size: 30
thread_pool.search.queue_size: 1500
```

16.1.2 scaling

scaling 线程池的线程数量是动态的，介于 core 和 max 参数之间变化。线程池的最小线程数为配置的 core 大小，随着请求的增加，当 core 数量的线程全都繁忙时，线程数逐渐增大到 max 数量。max 是线程池可拥有的线程数上限。当线程空闲时，线程数从 max 大小逐渐降低到 core 大小。

keep_alive 参数用来控制线程在线程池中的最长空闲时间。

例如：

```
thread_pool.warmer.core: 1
thread_pool.warmer.max: 8
thread_pool.warmer.keep_alive: 2m
```

16.1.3 direct

这种线程池对用户并不可见，当某个任务不需要在独立的线程执行，又想被线程池管理时，于是诞生了这种特殊类型的线程池：在调用者线程中执行任务。

16.1.4 fixed_auto_queue_size

与 fixed 类型的线程池相似，该线程池的线程数量为固定值，但是队列类型不一样。其队列大小根据利特尔法则（Little's Law）自动调整大小。该法则的详细信息可以参考 https://en.wikipedia.org/wiki/Little%27s_law。该线程池有以下参数可以调整：

- size，用于指定线程数量；
- queue_size，指定初始队列大小；
- min_queue_size，最小队列大小；
- max_queue_size，最大队列大小；
- auto_queue_frame_size，调整队列之前进行测量的操作数；
- target_response_time，一个时间值设置，用来指示任务的平均响应时间，如果任务经常高于这个时间，则线程池队列将被调小，以便拒绝任务。

该线程类型为实验性质，未来可能会移除。目前只有 search 线程池使用这种类型。

16.2 处理器设置

默认情况下，ES 自动探测处理器数量。各个线程池的大小基于这个数量进行初始化。在某些情况下，如果想手工指定处理器数量，则可以通过设置 processors 参数实现：

```
processors: 2
```

有以下几种场景是需要明确设置 processors 数量的：

（1）在同一台主机上运行多个 ES 实例，但希望每个实例的线程池只根据一部分 CPU 来设置，此时可以通过 processors 参数来设置处理器数量。例如，在 16 核的服务器上运行 2 个实

例，可以将 `processors` 设置为 8。请注意，在单台主机上运行多个实例，除了设置 `processors` 数量，还有许多更复杂的参数需要设置。例如，修改 GC 线程数，绑定进程到 CPU 等。

(2) 有时候自动探测出的处理器数量是错误的，在这种情况下，需要明确设置处理器数量。

要检查自动探测的处理器数量，可以使用节点信息 API 中的 `os` 字段来查看。

16.3 查看线程池

ES 提供了丰富的 API 查看线程池状态，在监控节点健康、排查问题时非常有用。

16.3.1 cat thread pool

该命令显示每个节点的线程池统计信息。默认情况下，所有线程池都返回活动、队列和被拒绝的统计信息。我们需要特别留意被拒绝的信息，例如，`bulk` 请求被拒绝意味着客户端写入失败。在正常情况下客户端应该捕获这种错误（错误码 429）并延迟重试，但有时客户端不一定对这种错误做了处理，导致写入集群的数据量低于预期值。

```
curl -X GET "localhost:9200/_cat/thread_pool"
```

返回信息如下：

node_name	name	active	queue	rejected
fc6s0S0	bulk	0	0	0
fc6s0S0	fetch_shard_started	0	0	0
fc6s0S0	fetch_shard_store	0	0	0
fc6s0S0	flush	0	0	0
fc6s0S0	force_merge	0	0	0
fc6s0S0	generic	0	0	0
fc6s0S0	get	0	0	0
fc6s0S0	index	0	0	0
fc6s0S0	listener	0	0	0
fc6s0S0	management	1	0	0
fc6s0S0	refresh	0	0	0
fc6s0S0	search	0	0	0
fc6s0S0	snapshot	0	0	0
fc6s0S0	warmer	0	0	0

`active` 表示当前正在执行任务的线程数量, `queue` 表示队列中等待处理的请求数量, `rejected` 表示由于队列已满, 请求被拒绝的次数。

对返回结果进行过滤等更多用法可参考官方手册: <https://www.elastic.co/guide/en/elasticsearch/reference/6.1/cat-thread-pool.html>。

16.3.2 nodes info

节点信息 API 可以返回每个线程池的类型和配置信息, 例如, 线程数量、队列大小等。

下面的第一条命令返回所有节点的信息, 第二条命令返回特定节点的信息。

```
curl -X GET "localhost:9200/_nodes"
curl -X GET "localhost:9200/_nodes/nodeId1,nodeId2"
```

节点信息 API 返回的信息非常大, 其中与线程池相关信息在 `thread_pool` 字段中, 选取部分信息如下:

```
"thread_pool" : {
  "force_merge" : {
    "type" : "fixed",
    "min" : 1,
    "max" : 1,
    "queue_size" : -1
  },
  "fetch_shard_started" : {
    "type" : "scaling",
    "min" : 1,
    "max" : 16,
    "keep_alive" : "5m",
    "queue_size" : -1
  }
}
```

该命令的完整信息可参考官方手册: <https://www.elastic.co/guide/en/elasticsearch/reference/6.1/cluster-nodes-info.html>。



16.3.3 nodes stats

stats API 返回集群中一个或全部节点的统计数据。

下面的第一条命令返回所有节点的统计数据，第二条命令返回特定节点的统计数据。

```
curl -X GET "localhost:9200/_nodes/stats"
curl -X GET "localhost:9200/_nodes/nodeId1,nodeId2/stats"
```

默认情况下，该 API 返回全部 indices、os、process、jvm、transport、http、fs、breaker 和 thread_pool 方面的统计数据。其中线程池相关的返回结果摘要如下：

```
"thread_pool" : {
  "bulk" : {
    "threads" : 0,
    "queue" : 0,
    "active" : 0,
    "rejected" : 0,
    "largest" : 0,
    "completed" : 0
  }
}
```

该命令的完整使用方式可参考官方手册：<https://www.elastic.co/guide/en/elasticsearch/reference/6.1/cluster-nodes-stats.html>。

16.3.4 nodes hot threads

该 API 返回集群中一个或全部节点的热点线程。

当发现节点进程占用 CPU 非常高时，想知道是哪些线程导致的，这些线程具体在执行什么操作，常规做法是通过 top 命令配合 jstack 来定位线程，现在 ES 提供了更便捷的方式，通过 hot threads API 可以直接返回这些信息。

下面的第一条命令返回所有节点的热点线程，第二条命令返回特定节点的热点线程。

```
curl -X GET "localhost:9200/_nodes/hot_threads"
curl -X GET "localhost:9200/_nodes/nodeId1,nodeId2/hot_threads"
```



该命令支持以下参数:

- threads, 返回的热点线程数, 默认为 3。
- interval, ES 对线程做两次检查, 来计算某个操作上花费时间的百分比, 此参数定义这个间隔时间。默认为 500 ms。
- type, 定义要检查的线程状态类型, 默认为 CPU。API 可以检查线程的 CPU 占用时间、阻塞 (block) 时间和等待 (wait) 时间。
- ignore_idle_threads, 如果设置为 true, 则空闲线程 (例如, 在套接字中等待, 或者从空队列中获取任务) 将被过滤。默认值为 true。

其返回信息的样例如下图所示。

```
::: {node1}{un-9UZ4PS8-K6hF59x1MWA}{bjk2C_6UShOYgMYKcBWKLQ}{node1.eshost}{10.10.13.15:9300}
Hot threads at 2018-06-06T03:51:46.567Z, interval=500ms, busiestThreads=3, ignoreIdleThreads=true:

82.2% (411.2ms out of 500ms) cpu usage by thread 'elasticsearch[node1][bulk][T#1]'
2/10 snapshots sharing following 37 elements
org.apache.lucene.index.DefaultIndexingChain.processField(DefaultIndexingChain.java:447)
org.apache.lucene.index.DefaultIndexingChain.processDocument(DefaultIndexingChain.java:403)
org.apache.lucene.index.DocumentsWriterPerThread.updateDocument(DocumentsWriterPerThread.java:232)
org.apache.lucene.index.DocumentsWriter.updateDocument(DocumentsWriter.java:478)
org.apache.lucene.index.IndexWriter.updateDocument(IndexWriter.java:1571)
org.apache.lucene.index.IndexWriter.addDocument(IndexWriter.java:1316)
org.elasticsearch.index.engine.InternalEngine.index(InternalEngine.java:663)
org.elasticsearch.index.engine.InternalEngine.indexIntoLucene(InternalEngine.java:607)
org.elasticsearch.index.engine.InternalEngine.index(InternalEngine.java:505)
org.elasticsearch.index.shard.IndexShard.index(IndexShard.java:556)
org.elasticsearch.index.shard.IndexShard.index(IndexShard.java:545)
org.elasticsearch.action.bulk.TransportShardBulkAction.executeIndexRequestOnPrimary(TransportShardBulkAction.java:491)
org.elasticsearch.action.bulk.TransportShardBulkAction.executeBulkItemRequest(TransportShardBulkAction.java:146)
org.elasticsearch.action.bulk.TransportShardBulkAction.shardOperationOnPrimary(TransportShardBulkAction.java:115)
org.elasticsearch.action.bulk.TransportShardBulkAction.shardOperationOnPrimary(TransportShardBulkAction.java:70)
```

返回信息中的第一行表明这个是哪个节点的信息, 以及这个节点的 IP 地址等。

```
::: {node1}{un-9UZ4PS8-K6hF59x1MWA}{bjk2C_6UShOYgMYKcBWKLQ}{node1.eshost}
{10.10.13.15:9300}
```

接下来列出哪个线程占用较多的 CPU, 以及 CPU 的占用比:

```
82.2% (411.2ms out of 500ms) cpu usage by thread 'elasticsearch[node1]
[bulk][T#1]'
```

最后是该线程的堆栈信息。

ES 中的线程池是基于对 Java 线程池的封装和扩展。我们先看一下 Java 线程池的结构和使用方式, 这些是 ES 内部线程原理的基础知识。

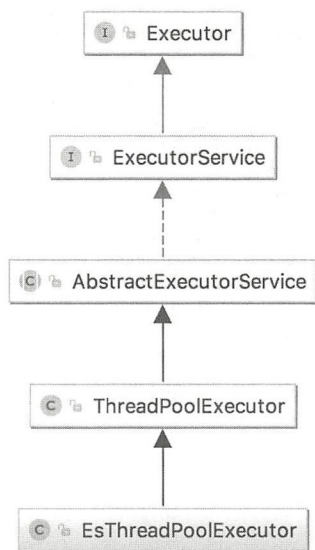


16.3.5 Java 的线程池结构

Java 内部的线程池称为 Executor 框架，几个基本的类概念如下：

- Runnable 定义一个要执行的任务。
- Executor 提供了 execute 方法，接受一个 Runnable 实例，用来执行一个任务。
- ExecutorService 是线程池的虚基类，继承自 Executor，提供了 shutdown、shutdownNow 等关闭线程池接口。
- ThreadPoolExecutor 线程池的具体实现。继承自 ExecutorService，维护线程创建、线程生命周期、任务队列等。
- EsThreadPoolExecutor 是 ES 对 ThreadPoolExecutor 的扩展实现。未来会增加一些统计信息。

这几个类的继承结构如下图所示。



我们以一个简单的例子来看看 Java 线程池的用法，ExecutorService 类用于保存创建的线程池实例，后续调用 execute 方法执行任务。在下面的例子中，任务类 TestRunnable 只是打印当前线程名称。

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolDemo{
```



```
public static void main(String[] args){
    //通过 Executors 构建一个固定大小的线程池，线程数量为 2，返回线程池实例
    ExecutorService executorService = Executors.newFixedThreadPool(2);
    //调用线程池的 execute 方法执行一个任务
    executorService.execute(new TestRunnable());
}

class TestRunnable implements Runnable{
    public void run(){
        System.out.println(Thread.currentThread().getName());
    }
}
```

ES 内部创建线程池时，返回类型同样是 `ExecutorService` 类。接下来我们通过构建过程来看 `ThreadPoolExecutor` 的结构，其构造函数如下：

```
public ThreadPoolExecutor(int corePoolSize,
    int maximumPoolSize,
    long keepAliveTime,
    TimeUnit unit,
    BlockingQueue<Runnable> workQueue,
    ThreadFactory threadFactory,
    RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```




几个重要参数的含义如下：

- `corePoolSize`，线程池大小；
- `maximumPoolSize`，最大线程数量；
- `keepAliveTime`，线程空闲回收时间；
- `BlockingQueue`，任务队列；
- `handler`，队列满，拒绝请求时的回调函数。

`ThreadPoolExecutor` 类是 Java 线程池的具体实现，是整个线程池中最重要的类，ES 基于这个类进行了一些扩展。

16.4 ES 的线程池实现

ES 中使用的线程池绝大部分封装在 `ThreadPool` 类中，个别独立线程池的实现在本章末尾讨论。除了个别情况，在 `ThreadPool` 类中，会创建各个模块要使用的全部线程池。本章开始所讨论的几种线程池就是在 `ThreadPool` 类中创建的。

`ThreadPool` 类创建各个线程池，要使用线程池的各个内部模块会引用 `ThreadPool` 类对象，通过其对外提供 `executor` 方法，根据线程池名称获取对应的线程池引用，进而执行某个任务。

`ThreadPool` 对外提供的重要方法如下表所示。

方 法	简 介
<code>executor</code>	通过线程池名称获取线程池对象： <code>ExecutorService</code>
<code>schedule</code>	在指定的线程池中延迟一定时间后执行一个任务，只执行一次，非周期性
<code>shutdown</code>	关闭线程池，不再接收新任务，待现有任务执行完毕才退出
<code>shutdownNow</code>	关闭线程池，不再接收新任务，尝试停止现有任务，并返回未执行的任务列表
<code>stats</code>	获取线程池状态

主要数据成员如下表所示。

成 员	简 介
<code>executors</code>	保存线程池名称与创建好的线程池的映射
<code>scheduler</code>	JDK 的 <code>ScheduledThreadPoolExecutor</code> 类对象，用于延迟执行任务

当某个模块要在新的线程中启动任务时，典型的使用方式如下：

```
threadPool.executor(ThreadPool.Names.SNAPSHOT).execute(() ->
    beginSnapshot(newState, newSnapshot, request.partial(), listener)
);
```



`threadPool.executor` 方法返回 `snapshot` 线程池 (`ExecutorService` 类) 的引用, 通过线程池的 `execute` 方法执行任务, 在本例中, 任务的 `Runnable` 是 `Lambda` 表达式定义的。

16.4.1 ThreadPool 类结构与初始化

`ThreadPool` 类对象在节点启动时初始化, 在 `Node` 类的构造函数中初始化 `ThreadPool` 类:

```
final ThreadPool threadPool = new ThreadPool(settings,
executorBuilders.toArray(new ExecutorBuilder[0]));
```

线程池对象构建完毕, 将这个类的引用传递给其他要使用线程池的模块:

```
final ResourceWatcherService resourceWatcherService = new
ResourceWatcherService(settings, threadPool);
```

线程池的名称在内部类 `Names` 中, 最好记住它们的名字, 有时需要通过 `jstack` 查看堆栈, `ES` 的堆栈非常长, 这就需要通过线程池的名称去查找关注的内容。

```
public static class Names {
    public static final String SAME = "same";
    public static final String GENERIC = "generic";
    public static final String LISTENER = "listener";
    public static final String GET = "get";
    public static final String INDEX = "index";
    public static final String BULK = "bulk";
    public static final String SEARCH = "search";
    public static final String MANAGEMENT = "management";
    public static final String FLUSH = "flush";
    public static final String REFRESH = "refresh";
    public static final String WARMER = "warmer";
    public static final String SNAPSHOT = "snapshot";
    public static final String FORCE_MERGE = "force_merge";
    public static final String FETCH_SHARD_STARTED = "fetch_shard_started";
    public static final String FETCH_SHARD_STORE = "fetch_shard_store";
}
```

线程池类型由枚举类型 `ThreadPoolType` 定义:



```
enum ThreadPoolType {
    DIRECT("direct"),
    FIXED("fixed"),
    FIXED_AUTO_QUEUE_SIZE("fixed_auto_queue_size"),
    SCALING("scaling");
}
```

在 `ThreadPool` 类构造函数中，全部的线程池被初始化：

```
public ThreadPool(final Settings settings, final ExecutorBuilder<?>...
customBuilders) {

    final Map<String, ExecutorBuilder> builders = new HashMap<>();
    builders.put(Names.GENERIC, new ScalingExecutorBuilder
(Names.GENERIC, 4, genericThreadPoolMax, TimeValue.timeValueSeconds(30)));
    builders.put(Names.INDEX, new FixedExecutorBuilder(settings,
Names.INDEX, availableProcessors, 200));
    //index/delete 操作与 bulk 使用同一个线程池
    builders.put(Names.BULK, new FixedExecutorBuilder(settings, Names.BULK,
availableProcessors, 200));
    builders.put(Names.GET, new FixedExecutorBuilder(settings, Names.GET,
availableProcessors, 1000));
    builders.put(Names.SEARCH, new AutoQueueAdjustingExecutorBuilder
(settings,
        Names.SEARCH, searchThreadPoolSize(availableProcessors),
1000, 1000, 1000, 2000));
    builders.put(Names.MANAGEMENT, new ScalingExecutorBuilder
(Names.MANAGEMENT, 1, 5, TimeValue.timeValueMinutes(5)));
    builders.put(Names.LISTENER, new FixedExecutorBuilder(settings,
Names.LISTENER, halfProcMaxAt10, -1));
    builders.put(Names.FLUSH, new ScalingExecutorBuilder(Names.FLUSH, 1,
halfProcMaxAt5, TimeValue.timeValueMinutes(5)));
    builders.put(Names.REFRESH, new ScalingExecutorBuilder
(Names.REFRESH, 1, halfProcMaxAt10, TimeValue.timeValueMinutes(5)));
    builders.put(Names.WARMER, new ScalingExecutorBuilder(Names.WARMER,
1, halfProcMaxAt5, TimeValue.timeValueMinutes(5)));
    builders.put(Names.SNAPSHOT, new ScalingExecutorBuilder
(Names.SNAPSHOT, 1, halfProcMaxAt5, TimeValue.timeValueMinutes(5)));
```



```
        builders.put (Names.FETCH_SHARD_STARTED, new ScalingExecutorBuilder
(Names.FETCH_SHARD_STARTED, 1, 2 * availableProcessors, TimeValue.timeValueMinutes
(5)));

        builders.put (Names.FORCE_MERGE, new FixedExecutorBuilder (settings,
Names.FORCE_MERGE, 1, -1));
        builders.put (Names.FETCH_SHARD_STORE, new ScalingExecutorBuilder
(Names.FETCH_SHARD_STORE, 1, 2 * availableProcessors, TimeValue.timeValueMinutes
(5)));
    }
```

这些线程池构建成功后，最终保存到一个 map 结构中，map 列表根据 builders 信息构建，将 SAME 线程池单独添加进去。

```
Map<String, ExecutorHolder> executors
```

当某个模块使用线程池时，通过线程池名称从这个 map 中获取对应的线程池。

```
public ExecutorService executor (String name) {
    final ExecutorHolder holder = executors.get (name);
    return holder.executor ();
}
```

map 中的值: ExecutorHolder 是 ThreadPool 的内部类，它只是简单封装了一下 ExecutorService。

```
class ExecutorHolder {
    private final ExecutorService executor;
    //Info 类主要是线程池名称、类型、队列大小、线程数量的 max 和 min、keepAlive 时间
    public final Info info;
}
```

16.4.2 fixed 类型线程池构建过程

FixedExecutorBuilder 类用于 fixed 类型的线程池构建，它的主要实现是通过 EsExecutors.newFixed 方法创建一个 ExecutorService。由于是 fixed 类型的线程池，因此 EsThreadPoolExecutor 传入的 corePoolSize 和 maximumPoolSize 的大小相同。

```
public static EsThreadPoolExecutor newFixed (String name, int size, int
queueCapacity, ThreadFactory threadFactory, ThreadContext contextHolder) {
```



```
//使用有限或无限大小的阻塞队列初始化线程池队列
BlockingQueue<Runnable> queue;
if (queueCapacity < 0) {
    queue = ConcurrentCollections.newBlockingQueue();
} else {
    queue = new SizeBlockingQueue<>(ConcurrentCollections.<Runnable>
newBlockingQueue(), queueCapacity);
}
//创建线程池
return new EsThreadPoolExecutor(name, size, size, 0, TimeUnit.MILLISECONDS,
queue, threadFactory, new EsAbortPolicy(), contextHolder);
}
```

16.4.3 scaling 类型线程池构建过程

ScalingExecutorBuilder 用于 scaling 类型线程池的构建,它的主要实现是通过 EsExecutors.newScaling 方法创建一个 ExecutorService, min 和 max 分别对应 corePoolSize 和 maximumPoolSize。

```
public static EsThreadPoolExecutor newScaling(String name, int min, int max,
long keepAliveTime, TimeUnit unit, ThreadFactory threadFactory, ThreadContext
contextHolder) {
    //创建线程队列
    ExecutorScalingQueue<Runnable> queue = new ExecutorScalingQueue<>();
    //min 为 corePoolSize, max 为 maximumPoolSize
    EsThreadPoolExecutor executor = new EsThreadPoolExecutor(name, min, max,
keepAliveTime, unit, queue, threadFactory, new ForceQueuePolicy(), contextHolder);
    queue.executor = executor;
    return executor;
}
```

16.4.4 direct 类型线程池构建过程

direct 类型的线程池没有通过 *ExecutorBuilder 类创建,而是通过 EsExecutors.newDirectExecutorService 方法直接创建的,该方法中直接返回一个定义好的简单的线程池 DIRECT_EXECUTOR_SERVICE。该线程池的实现如下,在 execute 方法中直接运行这个任务,因此任务在调用者所在线程中执行。


```

    private static final ExecutorService DIRECT_EXECUTOR_SERVICE = new
AbstractExecutorService() {
    //不支持关闭
    public void shutdown() {
        throw new UnsupportedOperationException();
    }
    //不支持中止
    public boolean awaitTermination(long timeout, TimeUnit unit) throws
InterruptedException {
        throw new UnsupportedOperationException();
    }

    //直接调用任务的运行方法，因此在调用者的线程中执行任务
    public void execute(Runnable command) {
        command.run();
    }

};

```

16.4.5 fixed_auto_queue_size 类型线程池构建过程

该类型的线程池通过 `AutoQueueAdjustingExecutorBuilder` 类构建，构建过程的主要实现是通过 `EsExecutors.newAutoQueueFixed` 方法创建一个 `ExecutorService`。该线程池的队列是一个大小可调整的队列，而 `QueueResizingEsThreadPoolExecutor` 继承自 `EsThreadPoolExecutor`，在它的基础上实现了动态调整队列大小的利特尔法则（Little's Law）。

```

public static EsThreadPoolExecutor newAutoQueueFixed(String name, int size,
int initialQueueCapacity, int minQueueSize,
    int maxQueueSize, int frameSize, TimeValue targetedResponseTime,
    ThreadFactory threadFactory, ThreadContext contextHolder) {
    //初始大小必须大于1
    if (initialQueueCapacity <= 0) {
        throw new IllegalArgumentException();
    }
    //初始化一个动态调整的队列
    ResizableBlockingQueue<Runnable> queue =
        new ResizableBlockingQueue<>(ConcurrentCollections.<Runnable>
newBlockingQueue(), initialQueueCapacity);

```

```

        //创建线程池
        return new QueueResizingEsThreadPoolExecutor(name, size, size, 0,
            TimeUnit.MILLISECONDS,
                queue, minQueueSize, maxQueueSize, TimedRunnable::new, frameSize,
                targetedResponseTime, threadFactory,
                new EsAbortPolicy(), contextHolder);
    }

```

16.5 其他线程池

除了 ThreadPool 中封装的各种线程池，ES 中还有一种支持优先级的线程池：PrioritizedEsThreadPoolExecutor，这个线程池同样继承自 EsThreadPoolExecutor 类。目前，只有主节点执行集群任务，以及从节点应用集群状态时使用该类型的线程池。

该线程池通过 EsExecutors.newSinglePrioritizing 方法构建，线程池有固定大小，线程数为 1。

```

    public static PrioritizedEsThreadPoolExecutor newSinglePrioritizing
        (String name, ThreadFactory threadFactory, ThreadContext contextHolder,
        ScheduledExecutorService timer) {
        //corePoolSize 和 maximumPoolSize 都为 1
        return new PrioritizedEsThreadPoolExecutor(name, 1, 1, 0L,
            TimeUnit.MILLISECONDS, threadFactory, contextHolder, timer);
    }

```

在 PrioritizedEsThreadPoolExecutor 类的构造函数中，会为线程池创建一个支持优先级的队列：PriorityBlockingQueue，该队列是 JDK 中的实现，初始大小为 11，最大为 Integer.MAX_VALUE - 8，基本是无限的。关于该队列的更多信息可以参考 JDK 手册：<https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/PriorityBlockingQueue.html>。

16.6 思考与总结

(1) 每种不同类型的线程池有各自不同的队列类型。scaling 类型的线程池动态维护线程池数量。fixed_auto_queue_size 与 fix 类型的线程池都有固定的线程数。

(2) ThreadPool 类在节点启动时初始化，然后将类的引用传递给其他模块，其他模块通过明确指定线程名称从 ThreadPool 类中获取对应的线程池，然后执行自定义的任务。

(3) 节点关闭时，对线程池模块先调用 shutdown，等待 10 秒后，执行 shutdownNow。因此线程池中的任务有机会执行完毕，但在超时会尝试终止线程池中的任务。

17 chapter

第 17 章

Shrink 原理分析

索引分片数量一般在模板中统一定义，在数据规模比较大的索引中，索引分片数一般也大一些，在笔者的集群中设置为 24。同时按天生成新的索引，使用别名关联。但是，并非每天的索引数据量都很大，小数据量的索引同样有较大的分片数。在 ES 中，主节点管理分片是很大的工作量，降低集群整体分片数量可以减少 `recovery` 时间，减小集群状态的大小。因此，可以使用 `Shrink API` 缩小索引分片数。当索引缩小完成后，源索引可以删除。

`Shrink API` 是 ES 5.0 之后提供的新功能，其可以缩小主分片数量。但其并不对源索引直接进行缩小操作，而是使用与源索引相同的配置创建一个新索引，仅降低分片数。由于添加新文档时使用对分片数量取余获取目的分片的关系，新索引的主分片数必须是源索引主分片数的因数。例如，8 个分片可以缩小到 4、2、1 个分片。如果源索引的分片数为素数，则目标索引的分片数只能为 1。

下面举一个例子来分析缩小过程。

17.1 准备源索引

创建索引：`my_source_index`，包括 5 个主分片和 1 个副分片，并写入几条测试数据

通过下面的命令，将索引标记为只读，且所有分片副本都迁移到名为 `node-idea` 的节点上。

注意，“所有分片副本”不指索引的全部分片，无论主分片还是副分片，任意一个就可以。分配器也不允许将主副分片分配到同一节点。

```
curl -XPUT 'localhost:9200/my_source_index/_settings?pretty' -H 'Content-Type: application/json' -d'
{
  "settings": {
    "index.routing.allocation.require._name": "node-idea",
    "index.blocks.write": true
  }
}
```

选项 `index.blocks.write` 设置为 `true` 来禁止对索引的写操作。但索引的 `metadata` 可以正常写。

17.2 缩小索引

待分片迁移完毕，我们就可以执行 `Shrink` 操作了：

```
curl -XPOST 'localhost:9200/my_source_index/_shrink/my_target_index?pretty' -H 'Content-Type: application/json' -d'
{
  "settings": {
    "index.number_of_replicas": 1,
    "index.number_of_shards": 1,
    "index.codec": "best_compression"
  },
  "aliases": {
    "my_search_indices": {}
  }
}
```

以上代码将创建含有一个主分片和一个副分片的目的索引 `my_target_index`。

17.3 Shrink 的工作原理

引用官方手册对 `Shrink` 工作过程的描述：

- 以相同配置创建目标索引，但是降低主分片数量。

- 从源索引的 Lucene 分段创建硬链接到目的索引。如果系统不支持硬链接，那么索引的所有分段都将复制到新索引，将会花费大量时间。
- 对目标索引执行恢复操作，就像一个关闭的索引重新打开时一样。

17.3.1 创建新索引

使用旧索引的配置创建新索引，只是减少主分片的数量，所有副本都迁移到同一个节点。显然，创建硬链接时，源文件和目标文件必须在同一台主机。

17.3.2 创建硬链接

从源索引到目的索引创建硬链接。如果操作系统不支持硬链接，则复制 Lucene 分段。

在 Linux 下通过 `strace` 命令跟踪硬链接创建过程：

```
strace -e trace=file -p {pid}
```

Linux 下的 `strace` 命令用于跟踪系统调用，`trace=file` 表示只跟踪与文件操作相关的系统调用，关于该命令的完整使用方式请求可参考 `man` 手册。在 `strace` 命令的输出结果中，我们能清晰看到内部过程：

```
link("/Volumes/idea/nodes/0/indices/JYglvWRnSqmNgA3E1CahZw/0/index/.1.cfe\0", "/Volumes/idea/nodes/0/indices/RvDP65d-QD-QTpwOCaLWOg/0/index/.0.cfs\0")
link("/Volumes/idea/nodes/0/indices/JYglvWRnSqmNgA3E1CahZw/1/index/.1.si\0", "/Volumes/idea/nodes/0/indices/RvDP65d-QD-QTpwOCaLWOg/0/index/.0.cfe\0")
link("/Volumes/idea/nodes/0/indices/JYglvWRnSqmNgA3E1CahZw/1/index/.1.si\0", "/Volumes/idea/nodes/0/indices/RvDP65d-QD-QTpwOCaLWOg/0/index/.0.si\0")
link("/Volumes/idea/nodes/0/indices/JYglvWRnSqmNgA3E1CahZw/2/index/.0.cfe\0", "/Volumes/idea/nodes/0/indices/RvDP65d-QD-QTpwOCaLWOg/0/index/.1.cfe\0")
link("/Volumes/idea/nodes/0/indices/JYglvWRnSqmNgA3E1CahZw/2/index/.0.cfs\0", "/Volumes/idea/nodes/0/indices/RvDP65d-QD-QTpwOCaLWOg/0/index/.1.cfs\0")
link("/Volumes/idea/nodes/0/indices/JYglvWRnSqmNgA3E1CahZw/2/index/.1.cfe\0", "/Volumes/idea/nodes/0/indices/RvDP65d-QD-QTpwOCaLWOg/0/index/.2.cfe\0")
link("/Volumes/idea/nodes/0/indices/JYglvWRnSqmNgA3E1CahZw/2/index/.1.cfs\0", "/Volumes/idea/nodes/0/indices/RvDP65d-QD-QTpwOCaLWOg/0/index/.2.cfs\0")
link("/Volumes/idea/nodes/0/indices/JYglvWRnSqmNgA3E1CahZw/3/index/.0.cfe\0", "/Volumes/idea/nodes/0/indices/RvDP65d-QD-QTpwOCaLWOg/0/index/.3.cfe\0")
link("/Volumes/idea/nodes/0/indices/JYglvWRnSqmNgA3E1CahZw/3/index/.0.si\0", "/Volumes/idea/nodes/0/indices/RvDP65d-QD-QTpwOCaLWOg/0/index/.3.si\0")
link("/Volumes/idea/nodes/0/indices/JYglvWRnSqmNgA3E1CahZw/3/index/.0.cfs\0", "/Volumes/idea/nodes/0/indices/RvDP65d-QD-QTpwOCaLWOg/0/index/.3.cfs\0")
link("/Volumes/idea/nodes/0/indices/JYglvWRnSqmNgA3E1CahZw/4/index/.0.cfe\0", "/Volumes/idea/nodes/0/indices/RvDP65d-QD-QTpwOCaLWOg/0/index/.4.cfe\0")
link("/Volumes/idea/nodes/0/indices/JYglvWRnSqmNgA3E1CahZw/4/index/.0.si\0", "/Volumes/idea/nodes/0/indices/RvDP65d-QD-QTpwOCaLWOg/0/index/.4.si\0")
link("/Volumes/idea/nodes/0/indices/JYglvWRnSqmNgA3E1CahZw/4/index/.0.cfs\0", "/Volumes/idea/nodes/0/indices/RvDP65d-QD-QTpwOCaLWOg/0/index/.4.cfs\0")
```

- `JYglvWRnSqmNgA3E1CahZw` 为源索引；
- `RvDP65d-QD-QTpwOCaLWOg` 为目的索引；
- `0.cfe`、`0.si`、`_0.cfs` 类型的文件为 Lucene 编号为 0 的 segment，编号依次类推。

链接过程：从源索引的 `shard[0]` 开始，遍历所有 `shard`，将所有 segment 链接到目的索引，目的索引的 segment 从 0 开始命名，依次递增。在本例中，由于源索引的 `shard[0]` 没有数据，因此从 `shard[1]` 开始链接。

为什么一定要硬链接，不使用软链接？

Linux 的文件系统由两部分组成(实际上任何文件系统的基本概念都相似):inode 和 block。block 用于存储用户数据, inode 用于记录元数据, 系统通过 inode 定位唯一的文件。

- 硬链接: 文件有相同的 inode 和 block。
- 软链接: 文件有独立的 inode 和 block, block 内容为目的文件路径名。

那么为什么一定要硬链接过去呢? 从本质上来说, 我们需要保证 Shrink 之后, 源索引和目的索引是完全独立的, 读写和删除都不应该互相影响。如果软链接过去, 删除源索引, 则目的索引的数据也会被删除, 硬链接则不会。满足下面条件时操作系统才真正删除文件:

文件被打开的 fd 数量为 0 且硬链接数量为 0。

使用硬链接, 删除源索引, 只是将文件的硬链接数量减 1, 删除源索引和目的索引中的任何一个, 都不影响另一个正常读写。

由于使用了硬链接, 也因为硬链接的特性带来一些限制: 不能交叉文件系统或分区进行硬链接的创建, 因为不同分区和文件系统有自己的 inode。

不过, 既然都是链接, Shrink 完成后, 修改源索引, 目的索引会变吗? 答案是不会。虽然链接到了源分段, Shrink 期间索引只读, 目标索引能看到的只有源索引的当前数据, Shrink 完成后, 由于 Lucene 中分段的不变性, “write once” 机制保证每个文件都不会被更新。源索引新写入的数据随着 refresh 会生成新分段, 而新分段没有链接, 在目标索引中是看不到的。如果源索引进行 merge, 对源分段执行删除时, 只是硬链接数量减 1, 目标索引仍然不受影响。因此, Shrink 完毕后最终的效果就是, 两个索引的数据看起来是完全独立的。

经过链接过程之后, 主分片已经就绪, 副分片还是空的, 通过 recovery 将主分片数据复制到副分片。下面看一下相关实现代码

17.3.3 硬链接过程源码分析

硬链接过程在目标索引 my_target_index 的恢复流程中, 入口为 IndexShard#startRecovery, 有下列几种类型的 recovery:

- **EXISTING_STORE**, 主分片从 translog 恢复;
- **PEER**, 副分片从主分片远程拉取;
- **SNAPSHOT**, 从快照中恢复;
- **LOCAL_SHARDS**, 从同一个节点的其他分片恢复 Shrink 使用这种恢复类型。

shrink index 时的恢复类型为 LOCAL_SHARDS, 执行 storeRecovery.recoverFromLocalShards。

在 `addIndices` 中, 调用 Lucene 中的 `org.apache.lucene.store.HardlinkCopyDirectoryWrapper` 实现硬链接。

`addIndices` 将整个源索引的全部 shard 链接到目标路径:

```
addIndices(RecoveryState.Index indexRecoveryStats, Directory target,
Directory... sources)
```

本例中源索引有 5 个分片, `sources` 值如下:

```
0 = "(store(mmapfs(/V/idea/nodes/0/indices/-Puacb8gSQG4UAvr-vNopQ/0/index)))"
1 = "(store(mmapfs(/V/idea/nodes/0/indices/-Puacb8gSQG4UAvr-vNopQ/1/index)))"
2 = "(store(mmapfs(/V/idea/nodes/0/indices/-Puacb8gSQG4UAvr-vNopQ/2/index)))"
3 = "(store(mmapfs(/V/idea/nodes/0/indices/-Puacb8gSQG4UAvr-vNopQ/3/index)))"
4 = "(store(mmapfs(/V/idea/nodes/0/indices/-Puacb8gSQG4UAvr-vNopQ/4/index)))"
```

`target` 值如下:

```
store(mmapfs(/Volumes/RamDisk/idea/nodes/0/indices/Dcfi3m9kTW2Dfc2zUjMOo
Q/0/index))
```

18 chapter

第 18 章 写入速度优化

在 ES 的默认设置下，是综合考虑数据可靠性、搜索实时性、写入速度等因素的。当离开默认设置、追求极致的写入速度时，很多是以牺牲可靠性和搜索实时性为代价的。有时候，业务上对数据可靠性和搜索实时性要求并不高，反而对写入速度要求很高，此时可以调整一些策略，最大化写入速度。

接下来的优化基于集群正常运行的前提下，如果是集群首次批量导入数据，则可以将副本数设置为 0，导入完毕再将副本数调整回去，这样副分片只需要复制，节省了构建索引过程。

综合来说，提升写入速度从以下几方面入手：

- 加大 translog flush 间隔，目的是降低 iops、writeblock。
- 加大 index refresh 间隔，除了降低 I/O，更重要的是降低了 segment merge 频率。
- 调整 bulk 请求。
- 优化磁盘间的任务均匀情况，将 shard 尽量均匀分布到物理主机的各个磁盘。
- 优化节点间的任务分布，将任务尽量均匀地发到各节点。
- 优化 Lucene 层建立索引的过程，目的是降低 CPU 占用率及 I/O，例如，禁用_all 字段。

18.1 translog flush 间隔调整

从 ES 2.x 开始，在默认设置下，translog 的持久化策略为：每个请求都“flush”。对应配置项如下：

```
index.translog.durability: request
```

这是影响 ES 写入速度的最大因素。但是只有这样，写操作才有可能可靠的。如果系统可以接受一定概率的数据丢失（例如，数据写入主分片成功，尚未复制到副分片时，主机断电。由于数据既没有刷到 Lucene，translog 也没有刷盘，恢复时 translog 中没有这个数据，数据丢失），则调整 translog 持久化策略为周期性和一定大小的时候“flush”，例如：

```
index.translog.durability: async
```

设置为 async 表示 translog 的刷盘策略按 sync_interval 配置指定的时间周期进行。

```
index.translog.sync_interval: 120s
```

加大 translog 刷盘间隔时间。默认为 5s，不可低于 100ms。

```
index.translog.flush_threshold_size: 1024mb
```

超过这个大小会导致 refresh 操作，产生新的 Lucene 分段。默认值为 512MB。

18.2 索引刷新间隔 refresh_interval

默认情况下索引的 refresh_interval 为 1 秒，这意味着数据写 1 秒后就可以被搜索到，每次索引的 refresh 会产生一个新的 Lucene 段，这会导致频繁的 segment merge 行为，如果不需要这么高的搜索实时性，应该降低索引 refresh 周期，例如：

```
index.refresh_interval: 120s
```

18.3 段合并优化

segment merge 操作对系统 I/O 和内存占用都比较高，从 ES 2.0 开始，merge 行为不再由 ES 控制，而是由 Lucene 控制，因此以下配置已被删除：

```
indices.store.throttle.type
indices.store.throttle.max_bytes_per_sec
index.store.throttle.type
index.store.throttle.max_bytes_per_sec
```

改为以下调整开关：

```
index.merge.scheduler.max_thread_count
```

```
index.merge.policy.*
```

最大线程数 `max_thread_count` 的默认值如下：

```
Math.max(1, Math.min(4, Runtime.getRuntime().availableProcessors() / 2))
```

以上是一个比较理想的值，如果只有一块硬盘并且非 SSD，则应该把它设置为 1，因为在旋转存储介质上并发写，由于寻址的原因，只会降低写入速度。

merge 策略 `index.merge.policy` 有三种：

- `tiered`（默认策略）；
- `log_byte_size`；
- `log_doc`。

每个策略的具体描述可以参考 `Mastering Elasticsearch`：https://doc.yonyoucloud.com/doc/mastering-elasticsearch/chapter-3/36_README.html。目前我们使用默认策略，但是对策略的参数进行了一些调整。

索引创建时合并策略就已确定，不能更改，但是可以动态更新策略参数，可以不做此项调整。如果堆栈经常有很多 merge，则可以尝试调整以下策略配置：

```
index.merge.policy.segments_per_tier
```

该属性指定了每层分段的数量，取值越小则最终 `segment` 越少，因此需要 merge 的操作更多，可以考虑适当增加此值。默认为 10，其应该大于等于 `index.merge.policy.max_merge_at_once`。

```
index.merge.policy.max_merged_segment
```

指定了单个 `segment` 的最大容量，默认为 5GB，可以考虑适当降低此值。

18.4 indexing buffer

`indexing buffer` 在为 doc 建立索引时使用，当缓冲满时会刷入磁盘，生成一个新的 `segment`，这是除 `refresh_interval` 刷新索引外，另一个生成新 `segment` 的机会。每个 `shard` 有自己的 `indexing buffer`，下面的这个 `buffer` 大小的配置需要除以这个节点上所有 `shard` 的数量：

```
indices.memory.index_buffer_size
```


默认为整个堆空间的 10%。

```
indices.memory.min_index_buffer_size
```

默认为 48MB。

```
indices.memory.max_index_buffer_size
```

默认为无限制。

在执行大量的索引操作时, `indices.memory.index_buffer_size` 的默认设置可能不够, 这和可用堆内存、单节点上的 `shard` 数量相关, 可以考虑适当增大该值。

18.5 使用 bulk 请求

批量写比一个索引请求只写单个文档的效率高得多, 但是要注意 `bulk` 请求的整体字节数不要太大, 太大的请求可能会给集群带来内存压力, 因此每个请求最好避免超过几十兆字节, 即使较大的请求看上去执行得更好。

18.5.1 bulk 线程池和队列

建立索引的过程属于计算密集型任务, 应该使用固定大小的线程池配置, 来不及处理的任务放入队列。线程池最大线程数量应配置为 CPU 核心数+1, 这也是 `bulk` 线程池的默认设置, 可以避免过多的上下文切换。队列大小可以适当增加, 但一定要严格控制大小, 过大的队列导致较高的 GC 压力, 并可能导致 FGC 频繁发生。

18.5.2 并发执行 bulk 请求

`bulk` 写请求是个长任务, 为了给系统增加足够的写入压力, 写入过程应该多个客户端、多线程地并行执行, 如果要验证系统的极限写入能力, 那么目标就是把 CPU 压满。磁盘 `util`、内存等一般都不是瓶颈。如果 CPU 没有压满, 则应该提高写入端的并发数量。但是要注意 `bulk` 线程池队列的 `reject` 情况, 出现 `reject` 代表 ES 的 `bulk` 队列已满, 客户端请求被拒绝, 此时客户端会收到 429 错误 (`TOO_MANY_REQUESTS`), 客户端对此的处理策略应该是延迟重试。不可忽略这个异常, 否则写入系统的数据会少于预期。即使客户端正确处理了 429 错误, 我们仍然应该尽量避免产生 `reject`。因此, 在评估极限的写入能力时, 客户端的极限写入并发量应该控

制在不产生 reject 前提下的最大值为宜。

18.6 磁盘间的任务均衡

如果部署方案是为 `path.data` 配置多个路径来使用多块磁盘, 则 ES 在分配 shard 时, 落到各磁盘上的 shard 可能并不均匀, 这种不均匀可能会导致某些磁盘繁忙, 利用率在较长时间内持续达到 100%。这种不均匀达到一定程度会对写入性能产生负面影响。

ES 在处理多路径时, 优先将 shard 分配到可用空间百分比最多的磁盘上, 因此短时间内创建的 shard 可能被集中分配到这个磁盘上, 即使可用空间是 99% 和 98% 的差别。后来 ES 在 2.x 版本中开始解决这个问题: 预估一下 shard 会使用的空间, 从磁盘可用空间中减去这部分, 直到现在 6.x 版也是这种处理方式。但是实现也存在一些问题:

从可用空间减去预估大小

这种机制只存在于一次索引创建的过程中, 下一次的索引创建, 磁盘可用空间并不是上次做完减法以后的结果。这也可以理解, 毕竟预估是不准的, 一直减下去空间很快就减没了。

但是最终的效果是, 这种机制并没有从根本上解决问题, 即使没有完美的解决方案, 这种机制的效果也不够好。

如果单一的机制不能解决所有的场景, 那么至少应该为不同场景准备多种选择。

为此, 我们为 ES 增加了两种策略。

- 简单轮询: 在系统初始阶段, 简单轮询的效果是最均匀的。
- 基于可用空间的动态加权轮询: 以可用空间作为权重, 在磁盘之间加权轮询。

18.7 节点间的任务均衡

为了节点间的任务尽量均衡, 数据写入客户端应该把 bulk 请求轮询发送到各个节点。

当使用 Java API 或 REST API 的 bulk 接口发送数据时, 客户端将会轮询发送到集群节点, 节点列表取决于:

- 使用 Java API 时, 当设置 `client.transport.sniff` 为 `true` (默认为 `false`) 时, 列表为所有数据节点, 否则节点列表为构建客户端对象时传入的节点列表。
- 使用 REST API 时, 列表为构建对象时添加进去的节点。

Java API 的 `TransportClient` 和 REST API 的 `RestClient` 都是线程安全的, 如果写入程序自己

创建线程池控制并发，则应该使用同一个 Client 对象。在此建议使用 REST API，Java API 会在未来的版本中废弃，REST API 有良好的版本兼容性好。理论上，Java API 在序列化上有性能优势，但是只有在吞吐量非常大时才值得考虑序列化的开销带来的影响，通常搜索并不是高吞吐量的业务。

要观察 bulk 请求在不同节点间的均衡性，可以通过 cat 接口观察 bulk 线程池和队列情况：

```
_cat/thread_pool
```

18.8 索引过程调整和优化

18.8.1 自动生成 doc ID

通过 ES 写入流程可以看出，写入 doc 时如果外部指定了 id，则 ES 会先尝试读取原来 doc 的版本号，以判断是否需要更新。这会涉及一次读取磁盘的操作，通过自动生成 doc ID 可以避免这个环节。

18.8.2 调整字段 Mappings

(1) 减少字段数量，对于不需要建立索引的字段，不写入 ES。

(2) 将不需要建立索引的字段 index 属性设置为 not_analyzed 或 no。对字段不分词，或者不索引，可以减少很多运算操作，降低 CPU 占用。尤其是 binary 类型，默认情况下占用 CPU 非常高，而这种类型进行分词通常没有什么意义。

(3) 减少字段内容长度，如果原始数据的大段内容无须全部建立索引，则可以尽量减少不必要的内容。

(4) 使用不同的分析器 (analyzer)，不同的分析器在索引过程中运算复杂度也有较大的差异。

18.8.3 调整 _source 字段

_source 字段用于存储 doc 原始数据，对于部分不需要存储的字段，可以通过 includes excludes 过滤，或者将 _source 禁用，一般用于索引和数据分离。

这样可以降低 I/O 的压力，不过实际场景中大多不会禁用 _source，而即使过滤掉某些字



段，对于写入速度的提升作用也不大，满负荷写入情况下，基本是 CPU 先跑满了，瓶颈在于 CPU。

18.8.4 禁用_all 字段

从 ES 6.0 开始，_all 字段默认为不启用，而在此前的版本中，_all 字段默认是开启的。_all 字段中包含所有字段分词后的关键词，作用是可以在搜索的时候不指定特定字段，从所有字段中检索。ES 6.0 默认禁用_all 字段主要有以下几点原因：

- 由于需要从其他的全部字段复制所有字段值，导致_all 字段占用非常大的空间。
- _all 字段有自己的分析器，在进行某些查询时（例如，同义词），结果不符合预期，因为没有匹配同一个分析器。
- 由于数据重复引起的额外建立索引的开销。
- 想要调试时，其内容不容易检查。
- 有些用户甚至不知道存在这个字段，导致了查询混乱。
- 有更好的替代方法。

关于这个问题的更多讨论可以参考 <https://github.com/elastic/elasticsearch/issues/19784>。

在 ES 6.0 之前的版本中，可以在 mapping 中将 enabled 设置为 false 来禁用_all 字段：

```
curl -X PUT "localhost:9200/my_index" -H 'Content-Type: application/json' -d '{
  "mappings": {
    "type_1": {
      "_all": {
        "enabled": false
      },
      "properties": {...}
    }
  }
}
```

禁用_all 字段可以明显降低对 CPU 和 I/O 的压力。



18.8.5 对 Analyzed 的字段禁用 Norms

Norms 用于在搜索时计算 doc 的评分，如果不需要评分，则可以将其禁用：

```
"title": {"type": "string", "norms": {"enabled": false}}
```

18.8.6 index_options 设置

index_options 用于控制在建立倒排索引过程中，哪些内容会被添加到倒排索引，例如，doc 数量、词频、positions、offsets 等信息，优化这些设置可以一定程度降低索引过程中的运算任务，节省 CPU 占用率。

不过在实际场景中，通常很难确定业务将来会不会用到这些信息，除非一开始方案就明确是这样设计的。

18.9 参考配置

下面是笔者的线上环境使用的全局模板和配置文件的部分内容，省略掉了节点名称、节点列表等基础配置字段，仅列出与本文相关内容。

从 ES 5.x 开始，索引级设置需要写在模板中，或者在创建索引时指定，我们把各个索引通用的配置写到了模板中，这个模板匹配全部的索引，并且具有最低的优先级，让用户定义的模板有更高的优先级，以覆盖这个模板中的配置。

```
{
  "template": "*",
  "order" : 0,
  "settings": {
    "index.merge.policy.max_merged_segment" : "2gb",
    "index.merge.policy.segments_per_tier" : "24",
    "index.number_of_replicas" : "1",
    "index.number_of_shards" : "24",
    "index.optimize_auto_generated_id" : "true",
    "index.refresh_interval" : "120s",
    "index.translog.durability" : "async",
    "index.translog.flush_threshold_size" : "1000mb",
    "index.translog.sync_interval" : "120s",
```




```
        "index.unassigned.node_left.delayed_timeout" : "5d"  
    }  
}
```

elasticsearch.yml 中的配置:

```
indices.memory.index_buffer_size: 30%
```

18.10 思考与总结

(1) 方法比结论重要。一个系统性问题往往是多种因素造成的,在处理集群的写入性能问题上,先将问题分解,在单台上进行压测,观察哪种系统资源达到极限,例如,CPU 或磁盘利用率、I/O block、线程切换、堆栈状态等。然后分析并调整参数,优化单台上的能力,先解决局部问题,在此基础上解决整体问题会容易得多。

(2) 可以使用更好的 CPU,或者使用 SSD,对写入性能提升明显。在我们的测试中,在相同条件下,E5 2650V4 比 E5 2430 v2 的写入速度高 60%左右。

(3) 在我们的压测环境中,写入速度稳定在平均单机每秒 3 万条以上,使用的测试数据:每个文档的字段数量为 10 个左右,文档大小约 100 字节,CPU 使用 E5 2430 v2。



19 chapter

第 19 章 搜索速度的优化

本章讨论搜索速度的优化、搜索速度与系统资源、数据索引方式、查询方式等多个方面，下面我们逐一讨论如何优化搜索速度。

19.1 为文件系统 cache 预留足够的内存

在一般情况下，应用程序的读写都会被操作系统“cache”（除了 direct 方式），cache 保存在系统物理内存中（线上应该禁用 swap），命中 cache 可以降低对磁盘的直接访问频率。搜索很依赖对系统 cache 的命中，如果某个请求需要从磁盘读取数据，则一定会产生相对较高的延迟。应该至少为系统 cache 预留一半的可用物理内存，更大的内存有更高的 cache 命中率。

19.2 使用更快的硬件

写入性能对 CPU 的性能更敏感，而搜索性能在一般情况下更多的是在于 I/O 能力，使用 SSD 会比旋转类存储介质好得多。尽量避免使用 NFS 等远程文件系统，如果 NFS 比本地存储慢 3 倍，则在搜索场景下响应速度可能会慢 10 倍左右。这可能是因为搜索请求有更多的随机访问。

如果搜索类型属于计算比较多，则可以考虑使用更快的 CPU。



19.3 文档模型

为了让搜索时的成本更低，文档应该合理建模。特别是应该避免 join 操作，嵌套 (nested) 会使查询慢几倍，父子 (parent-child) 关系可能使查询慢数百倍，因此，如果可以通过非规范化 (denormalizing) 文档来回答相同的问题，则可以显著地提高搜索速度。

19.4 预索引数据

还可以针对某些查询的模式来优化数据的索引方式。例如，如果所有文档都有一个 price 字段，并且大多数查询在一个固定的范围上运行 range 聚合，那么可以通过将范围“pre-indexing”到索引中并使用 terms 聚合来加快聚合速度。

例如，文档起初是这样的：

```
PUT index/type/1
{
  "designation": "spoon",
  "price": 13
}
```

采用如下的搜索方式：

```
GET index/_search
{
  "aggs": {
    "price_ranges": {
      "range": {
        "field": "price",
        "ranges": [
          { "to": 10 },
          { "from": 10, "to": 100 },
          { "from": 100 }
        ]
      }
    }
  }
}
```



那么我们的优化是，在建立索引时对文档进行富化，增加 `price_range` 字段，mapping 为 `keyword` 类型：

```
PUT index
{
  "mappings": {
    "type": {
      "properties": {
        "price_range": {
          "type": "keyword"
        }
      }
    }
  }
}
```

```
PUT index/type/1
{
  "designation": "spoon",
  "price": 13,
  "price_range": "10-100"
}
```

接下来，搜索请求可以聚合这个新字段，而不是在 `price` 字段上运行 `range` 聚合。

```
GET index/_search
{
  "aggs": {
    "price_ranges": {
      "terms": {
        "field": "price_range"
      }
    }
  }
}
```



19.5 字段映射

有些字段的内容是数值，但并不意味着其总是应该被映射为数值类型，例如，一些标识符，将它们映射为 keyword 可能会比 integer 或 long 更好。

19.6 避免使用脚本

一般来说，应该避免使用脚本。如果一定要用，则应该优先考虑 `painless` 和 `expressions`。

19.7 优化日期搜索

在使用日期范围检索时，使用 `now` 的查询通常不能缓存，因为匹配到的范围一直在变化。但是，从用户体验的角度来看，切换到一个完整的日期通常是可以接受的，这样可以更好地利用查询缓存。

例如，有下列查询：

```
PUT index/type/1
{
  "my_date": "2016-05-11T16:30:55.328Z"
}
```

```
GET index/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "range": {
          "my_date": {
            "gte": "now-1h",
            "lte": "now"
          }
        }
      }
    }
  }
}
```


可以替换成下面的查询方式：

```
GET index/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "range": {
          "my_date": {
            "gte": "now-1h/m",
            "lte": "now/m"
          }
        }
      }
    }
  }
}
```

在这个例子中，我们将日期四舍五入到分钟，因此如果当前时间是 16:31:29，那么 range 查询将匹配 my_date 字段的值在 15:31~16:31 之间的所有内容。如果几个用户同时运行一个包含此范围的查询，则查询缓存可以加快查询速度。用于舍入的时间间隔越长，查询缓存就越有帮助，但要注意，太高的舍入也可能损害用户体验。

为了能够利用查询缓存，可以很容易将范围分割成一个大的可缓存部分和一个小的不可缓存部分，如下所示。

```
GET index/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "bool": {
          "should": [
            {
              "range": {
                "my_date": {
                  "gte": "now-1h",
                  "lte": "now-1h/m"
                }
              }
            }
          ]
        }
      }
    }
  }
}
```


然后用别名关联，或者使用索引通配符。这样，可以每天选一个时间点对昨天的索引执行 force-merge、Shrink 等操作。

19.9 预热全局序号 (global ordinals)

全局序号是一种数据结构，用于在 keyword 字段上运行 terms 聚合。它用一个数值来代表字段中的字符串值，然后为每一数值分配一个 bucket。这需要一个对 global ordinals 和 bucket 的构建过程。默认情况下，它们被延迟构建，因为 ES 不知道哪些字段将用于 terms 聚合，哪些字段不会。可以通过配置映射在刷新 (refresh) 时告诉 ES 预先加载全局序号：

```
PUT index
{
  "mappings": {
    "type": {
      "properties": {
        "foo": {
          "type": "keyword",
          "eager_global_ordinals": true
        }
      }
    }
  }
}
```

19.10 execution hint

terms 聚合有两种不同的机制：

- 通过直接使用字段值来聚合每个桶的数据 (map)。
- 通过使用字段的全局序号并为每个全局序号分配一个 bucket (global_ordinals)。

ES 使用 global_ordinals 作为 keyword 字段的默认选项，它使用全局序号动态地分配 bucket，因此内存使用与聚合结果中的字段数量是线性关系。在大部分情况下，这种方式的速度很快。

当查询只会匹配少量文档时，可以考虑使用 map。默认情况下，map 只在脚本上运行聚合时使用，因为它们没有序号。

```
GET /_search
{
  "aggs" : {
    "tags" : {
      "terms" : {
        "field" : "tags",
        "execution_hint": "map"
      }
    }
  }
}
```

19.11 预热文件系统 cache

如果 ES 主机重启,则文件系统缓存将为空,此时搜索会比较慢。可以使用 `index.store.preload` 设置,通过指定文件扩展名,显式地告诉操作系统应该将哪些文件加载到内存中,

例如,配置到 `elasticsearch.yml` 文件中:

```
index.store.preload: ["nvd", "dvd"]
```

或者在索引创建时设置:

```
PUT /my_index
{
  "settings": {
    "index.store.preload": ["nvd", "dvd"]
  }
}
```

如果文件系统缓存不够大,则无法保存所有数据,那么为太多文件预加载数据到文件系统缓存中会使搜索速度变慢,应谨慎使用。

19.12 转换查询表达式

在组合查询中可以通过 `bool` 过滤器进行 `and`、`or` 和 `not` 的多个逻辑组合检索,这种组合查询中的表达式在下面的情况下可以做等价转换:

$$(A|B) \ \& \ (C|D) \ ==> \ (A\&C) \ | \ (A\&D) \ | \ (B\&C) \ | \ (B\&D)$$

19.13 调节搜索请求中的 `batched_reduce_size`

该字段是搜索请求中的一个参数。默认情况下，聚合操作在协调节点需要等所有的分片都取回结果后才执行，使用 `batched_reduce_size` 参数可以不等待全部分片返回结果，而是在指定数量的分片返回结果之后就可以先处理一部分（`reduce`）。这样可以避免协调节点在等待全部结果的过程中占用大量内存，避免极端情况下可能导致的 OOM。该字段的默认值为 512，从 ES 5.4 开始支持。

19.14 使用近似聚合

近似聚合以牺牲少量的精确度为代价，大幅提高了执行效率，降低了内存使用。近似聚合的使用方式可以参考官方手册：

- Percentiles Aggregation (<https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-metrics-percentile-aggregation.html>)。
- Cardinality Aggregation (<https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-metrics-cardinality-aggregation.html>)。

19.15 深度优先还是广度优先

ES 有两种不同的聚合方式：深度优先和广度优先。深度优先是默认设置，先构建完整的树，然后修剪无用节点。大多数情况下深度聚合都能正常工作，但是有些特殊的场景更适合广度优先，先执行第一层聚合，再继续下一层聚合之前会先做修剪，官方有一个例子可以参考：https://www.elastic.co/guide/cn/elasticsearch/guide/current/_preventing_combinatorial_explosions.html。

19.16 限制搜索请求的分片数

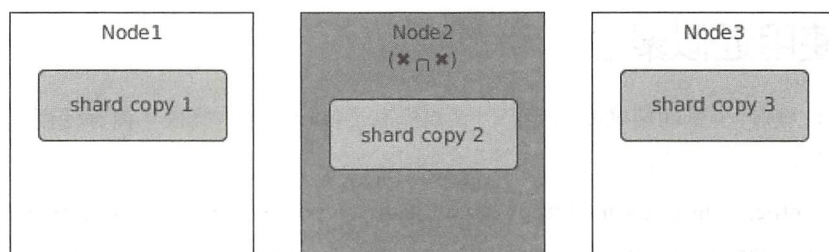
一个搜索请求涉及的分片数量越多，协调节点的 CPU 和内存压力就越大。默认情况下，ES 会拒绝超过 1000 个分片的搜索请求。我们应该更好地组织数据，让搜索请求的分片数更少。如果想调节这个值，则可以通过 `action.search.shard_count` 配置项进行修改。

虽然限制搜索的分片数并不能直接提升单个搜索请求的速度，但协调节点的压力会间接影响搜索速度，例如，占用更多内存会产生更多的 GC 压力，可能导致更多的 `stop-the-world` 时间等，因此间接影响了协调节点的性能，所以我们仍把它列作本章的一部分。

19.17 利用自适应副本选择（ARS）提升 ES 响应速度

为了充分利用计算资源和负载均衡，协调节点将搜索请求轮询转发到分片的每个副本，轮询策略是负载均衡过程中最简单的策略，任何一个负载均衡器都具备这种基础的策略，缺点是不考虑后端实际系统压力和健康水平。

例如，一个分片的三个副本分布在三个节点上，其中 Node2 可能因为长时间 GC、磁盘 I/O 过高、网络带宽跑满等原因处于忙碌状态，如下图所示。



如果搜索请求被转发到副本 2，则会看到相对于其他分片来说，副本 2 有更高的延迟。

- 分片副本 1：100ms。
- 分片副本 2（degraded）：1350ms。
- 分片副本 3：150ms。

由于副本 2 的高延迟，使得整个搜索请求产生长尾效应。

ES 希望这个过程足够智能，能够将请求路由到其他数据副本，直到该节点恢复到足以处理更多搜索请求的程度。在 ES 中，此过程称为“自适应副本选择”。

在实现过程中，ES 参考一篇名为 C3 的论文：*Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection* (<https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/suresh>)。这篇论文是为 Cassandra 写的，ES 基于这篇论文的思想做了调整以适合自己的场景。

ES 的 ARS 实现基于这样一个公式：对每个搜索请求，将分片的每个副本进行排序，以确定哪个最可能是转发请求的“最佳”副本。与轮询方式向分片的每个副本发送请求不同，ES 选择“最佳”副本并将请求路由到那里。ARS 公式为：

$$\Psi(s) = R(s) - 1/\bar{\mu}(s) + (\hat{q}(s))^3/\bar{\mu}(s)$$

$\hat{q}(s)$ 为：

$$\hat{q}(s) = 1 + (os(s) * n) + q(s)$$

每项含义如下：

- $os(s)$ ，节点未完成的搜索请求数；
- n ，系统中数据节点的数量；
- $R(s)$ ，响应时间的 EWMA（从协调节点上可以看到），单位为毫秒；
- $q(s)$ ，搜索线程池队列中等待任务数量的 EWMA；
- $\bar{\mu}(s)$ ，数据节点上的搜索服务时间的 EWMA，单位为毫秒。

关于 EWMA 的解释可参考 https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average。

通过这些信息我们大致可以评估出分片副本所在节点的压力和健康程度，这就可以让我们选出一个能够更快返回搜索请求的节点。在上面的例子中，请求将被转发到分片副本 1 或分片副本 3。

ARS 从 6.1 版本开始支持，但是默认关闭，可以通过下面的命令动态开启：

```
PUT /_cluster/settings
{
  "transient": {
    "cluster.routing.use_adaptive_replica_selection": true
  }
}
```

从 ES 7.0 开始，ARS 将默认开启。官方进行了多种场景的基准测试，包括某个数据节点处于高负载状态和非负载状态，测试使用 5 节点的集群，单个索引，5 个主分片，每个主分片有一个副分片。将搜索请求发送到单个协调节点。

没有模拟某个节点高负载的情况下（测试前节点都处于空闲），指标如下表所示。

指 标	不开启 ARS	开启 ARS	变化率（%）
吞吐量中位数（查询/秒）	95.7866	98.537	2.8
延迟中位数（毫秒）	1003.29	970.15	-3.3
第 90 个百分位的延迟（毫秒）	1339.69	1326.79	-0.9
第 99 个百分位的延迟（毫秒）	1648.34	1648.8	0.027

可见，即使集群的负载是均匀的，ARS 仍然可以改善吞吐量和响应延迟。

模拟单个节点处于高负载下的情况，指标如下表所示。

指 标	不开启 ARS	开启 ARS	变化率 (%)
吞吐量中位数 (查询/秒)	41.1558	87.8231	113.4
延迟中位数 (毫秒)	411.721	1007.22	144.6
第 90 个百分位的延迟 (毫秒)	5215.34	1839.46	-64.7
第 99 个百分位的延迟 (毫秒)	6181.48	2433.55	-60.6

使用 ARS，在某个数据节点处于高负载的情况下，吞吐量有了很大的提高。延迟中位数有所增加是预料之中的，为了绕开高负载的节点，稍微增加了无压力节点的负载，从而增加了延迟。

20 chapter

第 20 章 磁盘使用量优化

优化磁盘使用量与建立索引时的映射参数和索引元数据字段密切相关，在介绍具体的优化措施之前，我们先介绍这两方面的基础知识。

20.1 预备知识

20.1.1 元数据字段

每个文档都有与其相关的元数据，比如 `_index`、`_type` 和 `_id`。当创建映射类型时，可以定制其中一些元数据字段。下面列出了与本文相关的元数据字段，完整的介绍请参考官方手册：<https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-fields.html>。

- `_source`: 原始的 JSON 文档数据。
- `_all`: 索引所有其他字段值的一种通用字段，这个字段中包含了所有其他字段的值。允许在搜索的时候不指定特定的字段名，意味着“从全部字段中搜索”，例如：

```
http://localhost:9200/website/_search?q=keyword
```

`_all` 字段是一个全文字段，有自己的分析器。从 ES 6.0 开始该字段被禁用。之前的版本默认启用，但字段的 `store` 属性为 `false`，因此它不能被查询后取回显示。

20.1.2 索引映射参数

索引创建时可以设置很多映射参数，各种映射参数的详细说明可参考官方手册：<https://www.elastic.co/guide/en/elasticsearch/reference/master/mapping-params.html>，这里只介绍与本文相关的参数。

- **index**: 控制字段值是否被索引。它可以设置为 `true` 或 `false`，默认为 `true`。未被索引的字段不会被查询到，但是可以聚合。除非禁用 `doc_values`。
- **doc values**: 默认情况下，大多数字段都被索引，这使得它们可以搜索。倒排索引根据 `term` 找到文档列表，然后获取文档原始内容。但是排序和聚合，以及从脚本中访问某个字段值，需要不同的数据访问模式，它们不仅需要根据 `term` 找到文档，还要获取文档中字段的值。这些值需要单独存储。`doc_values` 就是用来存储这些字段值的。它是一种存储在磁盘上的列式存储，在文档索引时构建，这使得上述数据访问模式成为可能。它们以面向列的方式存储与 `_source` 相同的值，这使得排序和聚合效率更高。几乎所有字段类型都支持 `doc_values`，但被分析 (`analyzed`) 的字符串字段除外 (即 `text` 类型字符串)。`doc_values` 默认启用。
- **store**: 默认情况下，字段值会被索引使它们能搜索，但它们不会被存储 (`stored`)。意味着可以通过这个字段查询，但不能取回它的原始值。

但这没有关系。因为字段值已经是 `_source` 字段的一部分，它是被默认存储的。如果只想取回一个字段或少部分字段的值，而不是整个 `_source`，则可以通过 `source filtering` 达到目的。

在某些情况下，存储字段是有意义的。例如，如果有一个包含标题、日期和非常多的内容字段的文档，则可能希望只检索标题和日期，而不需要从大型 `_source` 字段中提取这些字段：

例如，我们创建一个索引：

```
PUT my_index
{
  "mappings": {
    "_doc": {
      "properties": {
        "title": {
          "type": "text",
          "store": true
        }
      }
    }
  }
}
```



```
        "type": "text"
    }
}
}
}
```

然后写入一条数据:

```
PUT my_index/_doc/1
{
  "title": "Some short title",
  "content": "A very long content field..."
}
```

下面的搜索将返回 title 字段的值:

```
GET my_index/_search
{
  "stored_fields": [ "title" ]
}
```

还有一种情况可能用到存储字段,就是不在 `_source` 中出现的字段(例如, `copy_to` 字段)。

`doc_values` 和存储字段 (`"stored":ture`) 都属于正排内容,两者的设计初衷不同。

`stored fields` 被设计为优化存储, `doc_values` 被设计为快速访问字段值。搜索可能会访问很多 `doc values` 中的字段,所以必须能够快速访问,我们将 `doc_values` 用于聚合、排序,以及脚本中。现在,ES 中的许多特性都会自动使用 `doc_values`。

另一方面,存储字段仅用于返回前几个最匹配文档的字段值,默认情况下 ES 只将其用于这种情况,解压存储字段,将其发送给客户端。为少量文档获取存储字段还好。它不能在查询的时候使用,否则会让查询变得非常慢。脚本中可以访问存储字段,但最好不要那么做。

20.2 优化措施

20.2.1 禁用对你来说不需要的特性

默认情况下,ES 为大多数的字段建立索引,并添加到 `doc_values`,以便使之可以被搜索和

聚合。但是有时候不需要通过某些字段过滤，例如，有一个名为 `foo` 的数值类型字段，需要运行直方图，但不需要在这个字段上过滤，那么可以不索引这个字段：

```
PUT index
{
  "mappings": {
    "type": {
      "properties": {
        "foo": {
          "type": "integer",
          "index": false
        }
      }
    }
  }
}
```

`text` 类型的字段会在索引中存储归一因子（`normalization factors`），以便对文档进行评分，如果只需要在文本字段上进行匹配，而不关心生成的得分，则可以配置 ES 不将 `norms` 写入索引：

```
PUT index
{
  "mappings": {
    "type": {
      "properties": {
        "foo": {
          "type": "text",
          "norms": false
        }
      }
    }
  }
}
```

`text` 类型的字段默认情况下也在索引中存储频率和位置。频率用于计算得分，位置用于执行短语（`phrase`）查询。如果不需要运行短语查询，则可以告诉 ES 不索引位置：

```
PUT index
{
  "mappings": {
    "type": {
      "properties": {
        "foo": {
          "type": "text",
          "index_options": "freqs"
        }
      }
    }
  }
}
```

在 `text` 类型的字段上，`index_options` 的默认值为 `positions`。`index_options` 参数用于控制添加到倒排索引中的信息。

`freqs` 文档编号和词频被索引，词频用于为搜索评分，重复出现的词条比只出现一次的词条评分更高。`positions` 文档编号、词频和位置被索引。位置被用于邻近查询（`proximity queries`）和短语查询（`phrase queries`）。

完整的 `index_options` 选项请参考官方手册：<https://www.elastic.co/guide/en/elasticsearch/reference/master/index-options.html>。

此外，如果也不关心评分，则可以将 ES 配置为只为每个 `term` 索引匹配的文档。仍然可以在这个字段上搜索，但是短语查询会出现错误，评分将假定在每个文档中只出现一次词汇。

```
PUT index
{
  "mappings": {
    "type": {
      "properties": {
        "foo": {
          "type": "text",
          "norms": false,
          "index_options": "freqs"
        }
      }
    }
  }
}
```

20.2.2 禁用 doc values

所有支持 doc value 的字段都默认启用了 doc value。如果确定不需要对字段进行排序或聚合，或者从脚本访问字段值，则可以禁用 doc value 以节省磁盘空间：

```
PUT my_index
{
  "mappings": {
    "_doc": {
      "properties": {
        "status_code": {
          "type": "keyword"
          "doc_values": false
        }
      }
    }
  }
}
```

20.2.3 不要使用默认的动态字符串映射

默认的动态字符串映射会把字符串类型的字段同时索引为 text 和 keyword。如果只需要其中之一，则显然是一种浪费。通常，id 字段只需作为 keyword 类型进行索引，而 body 字段只需作为 text 类型进行索引。

要禁用默认的动态字符串映射，则可以显式地指定字段类型，或者在动态模板中指定将字符串映射为 text 或 keyword。下例将字符串字段映射为 keyword：

```
PUT index
{
  "mappings": {
    "type": {
      "dynamic_templates": [
        {
          "strings": {
            "match_mapping_type": "string",
            "mapping": {
              "type": "keyword"
            }
          }
        }
      ]
    }
  }
}
```

```

    }
  ]
}
}
}

```

20.2.4 观察分片大小

较大的分片可以更有效地存储数据。为了增加分片大小，可以在创建索引的时候设置较少的主分片数量，或者使用 `shrink` API 来修改现有索引的主分片数量。但是较大的分片也有缺点，例如，较长的索引恢复时间。

20.2.5 禁用 `_source`

`_source` 字段存储文档的原始内容。如果不需要访问它，则可以将其禁用。但是，需要访问 `_source` 的 API 将无法使用，至少包括下列情况：

- `update`、`update_by_query`、`reindex`；
- 高亮搜索；
- 重建索引（包括更新 `mapping`、分词器，或者集群跨大版本升级可能会用到）；
- 调试聚合查询功能，需要对比原始数据。

20.2.6 使用 `best_compression`

`_source` 和设置为 `"store": true` 的字段占用磁盘空间都比较多。默认情况下，它们都是被压缩存储的。默认的压缩算法为 `LZ4`，可以通过使用 `best_compression` 来执行压缩比更高的算法：`DEFLATE`。但这会占用更多的 CPU 资源。

```

PUT index
{
  "settings": {
    "index": {
      "codec": "best_compression"
    }
  }
}

```


20.2.7 Fource Merge

一个 ES 索引由若干分片组成，一个分片有若干 Lucene 分段，较大的 Lucene 分段可以更有效地存储数据。

使用 `_forcemerge` API 来对分段执行合并操作，通常，我们将分段合并为一个单个的分段：`max_num_segments=1`。

20.2.8 Shrink Index

Shrink API 允许减少索引的分片数量，结合上面的 Force Merge API，可以显著减少索引的分片和 Lucene 分段数量。

20.2.9 数值类型长度够用就好

为数值类型选择的字段类型也可能会对磁盘使用空间产生较大影响，整型可以选择 `byte`、`short`、`integer` 或 `long`，浮点型可以选择 `scaled_float`、`float`、`double`、`half_float`，每个数据类型的字节长度是不同的，为业务选择够用的最小数据类型，可以节省磁盘空间。

20.2.10 使用索引排序来排列类似的文档

当 ES 存储 `_source` 时，它同时压缩多个文档以提高整体压缩比。例如，文档共享相同的字段名，或者它们共享一些字段值，特别是在具有低基数或 `zipfian` 分布（参考 https://en.wikipedia.org/wiki/Zipf%27s_law）的字段上。

默认情况下，文档按照添加到索引中的顺序压缩在一起。如果启用了索引排序，那么它们将按排序顺序压缩。对具有相似结构、字段和值的文档进行排序可以提高压缩比。

关于索引排序的详细内容请参考官方手册：<https://www.elastic.co/guide/en/elasticsearch/reference/master/index-modules-index-sorting.html>。

20.2.11 在文档中以相同的顺序放置字段

由于多个文档被压缩成块，如果字段总是以相同的顺序出现，那么在那些 `_source` 文档中可以找到更长的重复字符串的可能性更大。

20.3 测试数据

下面是在笔者的环境中，使用测试数据调整不同索引方式的测试结论。测试数据为单个文档十几个字段，大小为 800 字节左右。数据样本如下：

```
{ "status": "rst", "dst_mac": "xx:xx:xx:70:cb:75", "sip": "xxx.xxx.61.81",  
  "downlink_length": 7299, "down_payload":  
  "485454502f312e3120323030204f4b0d0a4461746500205765642c203033204d61792032303  
  1372030363a33323a343620474d540d0a53657276657200204170616368650d0a582d506f776  
  57265642d427900205048502f352e322e350d0a4361636865", "proto": "kerberos",  
  "dtime": "2017-11-21 12:19:45.358", "client_os": "linux2.2.x-3.x", "up_payload":  
  "504f5354202f716578717565727920485454502f312e310d0a557365722d4167656e7400205  
  06f73745f4d756c7469706172740d0a486f737400203130362e33382e3138342e3133360d0a4  
  1636365707400202a2f2a0d0a507261676d6100206e6f2d63", "serial_num": "XXJK/NOX+",  
  "server_os": "openbsd4", "summary": "110;2;1460;1380", "stime": "2017-11-21  
  11:43:05.247", "uplink_length": 1023, "dport": 43437, "sport": 54498, "dip":  
  "xxx.xxx.201.162", "src_mac": "xx:xx:xx:3c:30:67" }
```

在其他条件不变的情况下，调整单个参数，测试结果如下：

- 禁用 `_source`，空间占用量下降 30% 左右；
- 禁用 `doc values`，空间占用量下降 10% 左右；
- 压缩算法将 LZ4 改为 Deflate，空间占用量可以下降 15%~25%。

对于相似结构的数据，本文的测试结果可作一定参考，实际业务最好使用自己的样本数据进行压力测试以获取准确的结果。

21 chapter

第 21 章 综合应用实践

本章回答 ES 应该怎么用的问题。ES 被设计得简单易用，容易上手，如果只是把它当作黑盒来用，不了解内部原理，甚至没有一定的基础知识，当数据和节点规模达到一定程度的时候会面临许多问题。本章就重点问题给出使用和部署建议。

21.1 集群层

21.1.1 规划集群规模

在部署一个新集群时，应该根据多方面的情况评估需要多大的集群规模来支撑业务。这需要一些基础的测试数据，包括在特定的硬件下，特定业务数据样本的写入性能和搜索性能。然后根据具体业务情况来评估初始集群大小，这些信息包括：

- 数据总量，每天的增量；
- 查询类型和搜索并发，QPS；
- SLA 级别。

另一方面，需要控制最大集群规模和数据总量，参考下列两个限制条件：

- 节点总数不应该太多，一般来说，最大集群规模最好控制在 100 个节点左右。我们曾经测试过上千个节点集群，在这种规模下，节点间的连接数和通信量倍增，主节点管理压力比较大。

- 单个分片不要超过 50GB，最大集群分片总数控制在几十万的级别。太多分片同样增加了主节点的管理负担，而且集群重启恢复时间会很长。

建议为集群配置较好的硬件，而不是普通的 PC，搜索对 CPU、内存、磁盘的性能要求都很高，要达到比较低的延迟就需要较好的硬件资源。另外，如果使用不同配置的服务器混合部署，则搜索速度可能会取决于最慢的那个节点，产生长尾效应。

21.1.2 单节点还是多节点部署

ES 不建议为 JVM 配置超过 32GB 的内存，超过 32GB 时，Java 内存指针压缩失效，浪费一些内存，降低了 CPU 性能，GC 压力也较大。因此推荐设置为 31GB：

```
-Xmx31g -Xms31g
```

确保堆内存最小值（Xms）与最大值（Xmx）大小相同，防止程序在运行时动态改变堆内存大小，这是很耗系统资源的过程。

当物理主机内存存在 64GB 以上，并且拥有多个数据盘，不做 raid 的情况下，部署 ES 节点时有多种选择：

（1）部署单个节点，JVM 内存配置不超过 32GB，配置全部数据盘。这种部署模式的缺点是多余的物理内存只能被 cache 使用，而且只要存在一个坏盘，节点重启会无法启动。

（2）部署单个节点，JVM 内存配置超过 32GB，配置全部数据盘。接受指针压缩失效和更长时间的 GC 等负面影响。

（3）有多少个数据盘就部署多少个节点，每个节点配置单个数据路径。优点是可以统一配置，缺点是节点数较多，集群管理负担大，只适用于集群规模较小的场景。

（4）使用内存大小除以 64GB 来确定要部署的节点数，每个节点配置一部分数据盘，优点是利用率最高，缺点是部署复杂。

官方的建议是方案 4，但是为了管理和维护的简便，也可以使用方案 1 和 3。这两种部署模式在我们的集群中都在使用，集群规模较小时可以考虑使用多节点方式部署，例如，只有 3 台物理机。当集群规模较大时，建议单节点方式部署，例如，物理机达到 100 台以上。

21.1.3 移除节点

当由于坏盘、维护等故障需要下线一个节点时，我们需要先将该节点的数据迁移，这可以通过分配过滤器实现。例如，我们将 node-1 下线：

```
PUT _cluster/settings
{
  "transient" : {
    "cluster.routing.allocation.exclude._name" : "node-1"
  }
}
```

执行命令后，分片开始迁移，我们可以通过 `_cat/shard` API 来查看该节点的分片是否迁移完毕。当节点维护完毕，重新上线之后，需要取消排除设置，以便后续的分片可以分配到 `node-1` 节点上。

```
PUT _cluster/settings
{
  "transient" : {
    "cluster.routing.allocation.exclude._name" : ""
  }
}
```

完整的分配过滤器使用方式请参考官方手册：<https://www.elastic.co/guide/en/elasticsearch/reference/current/allocation-filtering.html>。

21.1.4 独立部署主节点

将主节点和数据节点分离部署最大的好处是 Master 切换过程可以迅速完成，有机会跳过 gateway 和分片重新分配的过程。例如，有 3 台具备 Master 资格的节点独立部署，然后关闭当前活跃的主节点，新主当选后由于内存中持有最新的集群状态，因此可以跳过 gateway 的恢复过程，并且由于主节点没有存储数据，所以旧的 Master 离线不会产生未分配状态的分片。新主当选后集群状态可以迅速变为 Green。

21.2 节点层

21.2.1 控制线程池的队列大小

不要为 bulk 和 search 分配过大的队列，队列并非越大越好，队列缓存的数据越多，GC 压力越大，默认的队列大小基本够用了，即使在压力测试的场景中，默认队列大小也足以支持。

除非在一些特别的情况下，例如，每个请求的数据量都非常小，可能需要增加队列大小。但是我们推荐写数据时组合较大的 bulk 请求。

21.2.2 为系统 cache 保留一半物理内存

搜索操作很依赖对系统 cache 的命中，标准的建议是把 50% 的可用内存作为 ES 的堆内存，为 Lucene 保留剩下的 50%，用作系统 cache。

21.3 系统层

21.3.1 关闭 swap

在个人 PC 上，交换分区或许有用，如果物理内存不够，则交换分区可以让系统缓慢运行。但是在服务器系统上，无论物理内存多么小，哪怕只有 1GB，都应该关闭交换分区。当服务程序在交换分区上缓慢运行时，往往会产生更多不可预期的错误，因此当一个申请内存的操作如果真的遇到物理内存不足时，宁可让它直接失败。

一般在安装操作系统的时候直接关闭交换分区，或者通过 `swaponoff` 命令来关闭。

21.3.2 配置 Linux OOM Killer

现在讨论的 OOM 并非 JVM 的 OOM，而是 Linux 操作系统的 OOM。在 Linux 下，进程申请的内存并不会立刻为进程分配真实大小的内存，因为进程申请的内存不一定全部使用，内核在利用这些空闲内存时采取过度分配的策略，假如物理内存为 1GB，则两个进程都可以申请 1GB 的内存，这超过了系统的实际内存大小。当应用程序实际消耗完内存的时候，怎么办？系统需要“杀掉”一些进程来保障系统正常运行。这就触发了 OOM Killer，通过一些策略给每个进程打分，根据分值高低决定“杀掉”哪些进程。默认情况下，占用内存最多的进程被“杀掉”。

如果 ES 与其他服务混合部署，当系统产生 OOM 的时候，ES 有可能会无辜被“杀”。为了避免这种情况，我们可以在用户态调节一些进程参数来让某些进程不容易被 OOM Killer “杀掉”。例如，我们不希望 ES 进程被“杀”，可以设置进程的 `oom_score_adj` 参数为 -17（越小越不容易被杀）：

```
$jps
1849 Elasticsearch
```

```
$cat /proc/1849/oom_score_adj
0

$sudo echo -17 > /proc/1849/oom_score_adj
```

可以将这个信息写到 ES 的启动脚本中自动执行。

21.3.3 优化内核参数

在生产环境上，我们可以根据自己的场景调节内核参数，让搜索服务更有效率地运行。例如，ES 集群中的节点一般处于同一个子网，也就是在同一个局域网，Linux 默认的 TCP 选项不一定完全合适，因为它需要考虑在互联网上传输时可能出现的更大的延迟和丢包率。因此我们可以调节一些 TCP 选项，让 TCP 协议在局域网上更高效。

调节内核参数可以通过两种方式：

(1) 临时设置，系统重启后失效。通过 `sysctl -w` 来设置，例如：`sysctl -w net.ipv4.tcp_timestamps=1`，命令执行后该参数立即生效。

(2) 永久设置，将参数写入配置文件 `/etc/sysctl.conf`，然后执行 `sysctl -p` 使其生效。

可以通过 `sysctl -A` 配合 `grep` 查看某个参数的当前值。

下面给出一些比较通用的内核参数设置建议，这些参数的默认值以 CentOS7.2 为参考，在其他系统上可能会有些差异。

1. TCP 相关参数

```
net.ipv4.tcp_syn_retries
```

默认值为 6，参考值为 2。主机作为客户端，对外发起 TCP 连接时，即三次握手的第一步，内核发送 SYN 报文的重试次数，超过这个次数后放弃连接。内网环境通信良好，因此可以适度降低此值。

```
net.ipv4.tcp_synack_retries
```

默认值为 5，参考值为 2。主机作为服务端，接受 TCP 连接时，在三次握手的第二步，向客户端发送 SYN+ACK 报文的重试次数，超过这个次数后放弃连接。内网环境中可适度降低此值。

```
net.ipv4.tcp_timestamps
```

默认值为 1，参考值为 1。是否开启时间戳，开启后可以更精确地计算 RTT，一些其他特性

也依赖时间戳字段。

```
net.ipv4.tcp_tw_reuse
```

默认值为 0，建议值为 1。是否允许将处于 TIME_WAIT 状态的 socket 用于新的 TCP 连接。这对于降低 TIME_WAIT 数据很有效。该参数只有在开启 tcp_timestamps 的情况下才会生效。

```
net.ipv4.tcp_tw_recycle
```

默认值为 0，参考值为 0。是否开启 TIME_WAIT 套接字的快速回收，这是比 tcp_tw_reuse 更激进的一种方式，它同样依赖 tcp_timestamps 选项。强烈建议不要开启 tcp_tw_recycle，原因有两点，一是 TIME_WAIT 是十分必要的状态，避免关闭中的连接与新建连接之间的数据混淆，二是 tcp_tw_recycle 选项在 NAT 环境下会导致一些新建连接被拒绝，因为 NAT 下每个主机存在时差，这体现在套接字中的时间戳字段，服务端会发现某个 IP 上的本应递增的时间戳出现降低的情况，时间戳相对降低的报文将被丢弃。

```
net.core.somaxconn
```

默认值为 128，参考值为 2048。定义了系统中每一个端口上最大的监听队列的长度。当服务端监听了某个端口时，操作系统内部完成对客户端连接请求的三次握手。这些已建立的连接存储在一个队列中，等待 accept 调用取走。本选项就是定义这个队列的长度。该队列实际大小取决于 listen 调用传入的第二个参数：backlog 和本选项的最小值：min(backlog,somaxconn)。ES 需要建立许多连接，当集群节点数比较大，集群完全重启时可能会在瞬间建立大量连接，默认的连接队列长度可能不够用，因此适当提高此值。

```
net.ipv4.tcp_max_syn_backlog
```

默认值为 128，参考值为 8192。内核会服务端的连接建立两个队列：

- 已完成三次握手，连接已建立，等待 accept 的队列，全局长度由 somaxconn 定义。
- 三次握手执行到第二步，等待客户端返回 ACK，这些未完成的连接单独放到一个队列中，由 tcp_max_syn_backlog 定义队列大小。

由于可能会有较多的连接数，我们适度增加“未完成连接”的队列大小。

```
net.ipv4.tcp_max_tw_buckets
```

默认值为 4096，参考值为 180000。定义系统同时保持 TIME_WAIT 套接字的最大数量，如果超过这个数，则 TIME_WAIT 套接字将立刻被清除并打印警告信息。如果系统被 TIME_WAIT

过多问题困扰,则可以调节 `tcp_max_tw_buckets`、`tcp_tw_reuse`、`tcp_timestamps` 三个选项来缓解。`TIME_WAIT` 状态产生在 TCP 会话关闭时主动关闭的一端,如果想从根本上解决问题,则让客户端主动关闭连接,而非服务端。

```
net.ipv4.tcp_max_orphans
```

默认值为 4096,参考值为 262144。定义最大孤儿套接字(未附加到任何用户文件句柄的套接字)数量。如果孤儿套接字数量超过此值,则这些连接立即“reset”,并显示警告信息。该值可以简单地抵御 DOS 攻击,但不能通过降低此值来抵御 DOS。为了应对高负载,应该提高此值。

2. TCP 的接收窗口 (RWND)

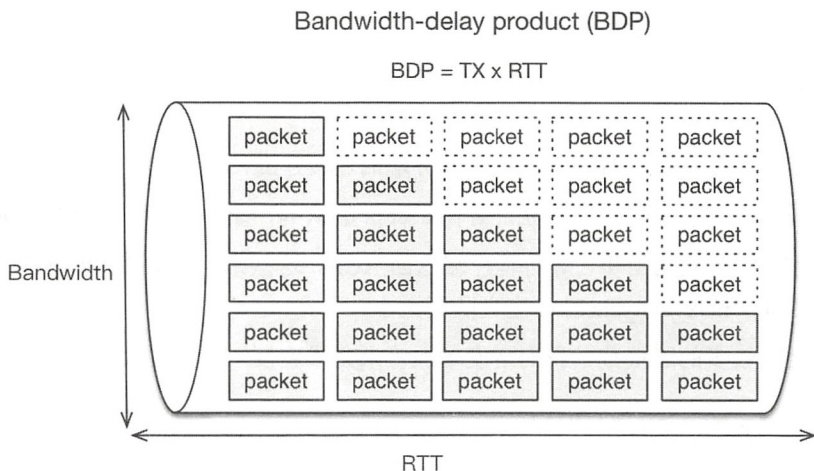
TCP 采用两个基本原则决定何时发送及发送多少数据:

- 流量控制,为了确保接收者可以接收数据。
- 拥塞控制,为了管理网络带宽。

流量控制通过在接收方指定接收窗口大小来实现。接收窗口用于接收端告诉发送端,自己还有多大的缓冲区可以接收数据。发送端参考这个值来发送数据,就不会导致客户端处理不过来。

接收窗口的大小可以通过内核参数来调整,其理想值是 BDP (bandwidth-delay product): 服务端可以发出的未被客户端确认的数据量,也就是在网络上缓存的数据量。

网络连接通常以管道为模型,BDP 是带宽与 RTT 的乘积,表示需要多少数据填充管道。下图展示了 BDP 的基本概念。



例如,在千兆的网络上,RTT 为 10 毫秒,那么 $BDP = (1000/8) \times 0.01s = 1.25MB$ 。在这种

情况下, 如果想最大限度地提升 TCP 吞吐量, 则 RWND 大小不应小于 1.25MB。

可以通过下面的选项调整 RWND:

```
net.ipv4.tcp_rmem = <MIN> <DEFAULT> <MAX>
```

默认情况下, 系统会在最大值和最小值之间自动调整缓冲区大小, 是否自动调整通过 `tcp_moderate_rcvbuf` 选项来决定。在开启缓冲自动调整的情况下, 可以把最大值设置为 BDP。

TCP 使用 2 个字节记录窗口大小, 因此最大值为 64KB, 如果超过这个值, 则需要使用 `tcp_window_scaling` 机制, 通过下面的设置开启 (默认启用):

```
net.ipv4.tcp_window_scaling = 1
```

RWND 和 CWND 可能是让系统达到最大吞吐量的两个限制因素, 接下来我们讨论 CWND。

3. TCP 的拥塞窗口 (CWND)

TCP 的滑动窗口机制依据接收端的能力来进行流控, 并不能感知网络延迟等网络因素。拥塞控制机制会评估网络能承受的负荷, 避免过量数据发送到网络中, 拥塞程度会涉及主机、路由器等网络上的所有因素。

拥塞控制由发送方实现, 发送方会将它传输的数据量限制为 CWND 和 RWND 的最小值。CWND 会随着时间和对端的 ACK 增长, 如果检测到网络拥塞, 则缩小 CWND。拥塞控制主要有 4 种算法: 慢启动、拥塞避免、快速重传和快速恢复。

慢启动的意思是对刚建立的连接, 开始发送数据时, 一点点提速, 而不是一下子使用很大的带宽。慢启动是指数上升的过程, 直到 $CWND \geq ssthresh$, 进入拥塞避免算法。

本节我们讨论的问题就是调节初始拥塞窗口 (INITCWND) 的大小。适度增加 INITCWND 可以降低 HTTP 响应延迟, 可以参考 Google 的论文: *An Argument for Increasing TCP's Initial Congestion Window*。

例如, HTTP 要返回的内容为 20K, MSS 的大小为 1460, 整个内容需要传送 15 个 MSS。当 INITCWND 为 3 时, 服务端先发送 3 个 MSS, $1460 \times 3 = 4380$ 字节, 待客户端 ACK 后, 根据指数增加算法, 第二次发送 9 个 MSS, $1460 \times 9 = 13140$ 字节, 第三次发送 3 个 MSS 的剩余字节。整个传输过程经过了 3 次 RTT。如果 INITCWND 设置为 15, 则只需要一次 RTT 就可以完成传输。

在 Linux 2.6.39 版本之前, INITCWND 值根据 MSS 值来计算, 参考 RFC3390: <http://www.rfc-editor.org/rfc/rfc3390.txt>。以太网的 MSS 大小一般是 1460, 因此 INITCWND 为 3, 这个初始值比较小。从 2.6.39 版本及之后, 采取了 Google 的建议, 把 INITCWND 调到了 10, 参考:

https://kernelnewbies.org/Linux_2_6_39#head-1d11935223b203d28a660417627514973de4e218。

如果系统上的 INITCWND 低于 10，可以使用 `ip` 命令调整。调节和测试 INITCWND 的具体方法可以参考：<https://www.cdnplanet.com/blog/tune-tcp-initcwnd-for-optimum-performance/>。

4. vm 相关参数

文件的读和写操作都会经过操作系统的 cache，读缓存是比较简单的，而写缓存相对复杂。在一般情况下，写文件的数据先到系统缓存（page cache），再由系统定期异步地刷入磁盘，这些存储于 page cache 中尚未刷盘的数据称为脏数据（或者脏页，dirty page）。写缓存可以提升 I/O 速度，但存在数据丢失的风险。例如，在尚未刷盘的时候主机断电。

系统当前 page cache 信息可以通过 `/proc/meminfo` 文件查看。下面我们讨论一下写缓存的细节和控制策略。

从 page cache 刷到磁盘有以下三种时机：

- 可用物理内存低于特定阈值时，为了给系统腾出空闲内存；
- 脏页驻留时间超过特定阈值时，为了避免脏页无限期驻留内存；
- 被用户的 `sync()` 或 `fsync()` 触发。

由系统执行的刷盘有两种写入策略：

- 异步执行刷盘，不阻塞用户 I/O；
- 同步执行刷盘，用户 I/O 被阻塞，直到脏页低于某个阈值。

在一般情况下，系统先执行第一种策略，当脏页数据量过大，异步执行来不及完成刷盘时，切换到同步方式。我们可以通过内核参数调整脏数据的刷盘阈值：

- `vm.dirty_background_ratio`，默认值为 10。该参数定义了一个百分比。当内存中的脏数据超过这个百分比后，系统使用异步方式刷盘。
- `vm.dirty_ratio`，默认值为 30。同样定义了一个百分比，当内存中的脏数据超过这个百分比后，系统使用同步方式刷盘，写请求被阻塞，直到脏数据低于 `dirty_ratio`。如果还高于 `dirty_background_ratio`，则切换到异步方式刷盘。因此 `dirty_ratio` 应高于 `dirty_background_ratio`。

除了通过百分比控制，还可以指定字节大小，类似的参数有：

```
dirty_background_bytes
dirty_bytes
```

- `vm.dirty_expire_centisecs`，默认值为 3000（30 秒），单位为百分之 1 秒，定义脏数据的过期时间，超过这个时间后，脏数据被异步刷盘。

- `vm.dirty_writeback_centisecs`，默认值为 500（5 秒），单位为百分之 1 秒，系统周期性地启动线程来检查是否需要刷盘，该选项定义这个间隔时间。

可以通过下面的命令查看系统当前的脏页数量：

```
cat /proc/vmstat | egrep "dirty|writeback"
nr_dirty 951
nr_writeback 0
nr_writeback_temp 0
```

输出显示有 951 个脏页等待写到磁盘。默认情况下每页大小为 4KB。另外，也可以在 `/proc/meminfo` 文件中看到这些信息。

如果数据安全性要求没有那么高，想要多“cache”一些数据，让读取更容易命中，则可以增加脏数据占比和过期时间：

```
vm.dirty_background_ratio = 30
vm.dirty_ratio = 60
vm.dirty_expire_centisecs = 6000
```

反之则可以降低它们。如果只希望写入过程不要被系统的同步刷盘策略影响，则可以让系统多容纳脏数据，但早一些触发异步刷盘。这样也可以让 I/O 更平滑：

```
vm.dirty_background_ratio = 5
vm.dirty_ratio = 60
```

5. 禁用透明大页（Transparent Hugepages）

透明大页是 Linux 的一个内核特性，它通过更有效地使用处理器的内存映射硬件来提高性能，默认情况下是启用的。禁用透明大页能略微提升程序性能，但是也可能对程序产生负面影响，甚至是严重的内存泄漏。为了避免这些问题，我们应该禁用它（许多项目都建议禁用透明大页，例如，MongoDB、Oracle）。可以通过下面的命令检查其是否开启：

```
cat /sys/kernel/mm/transparent_hugepage/enabled
[always] madvise never
```

`always` 代表开启，通过下面的命令将其禁用（系统重启后失效）：

```
echo never | sudo tee /sys/kernel/mm/transparent_hugepage/enabled
```

关于透明大页对应用程序的具体影响可以参考一篇分析文章：<https://blog.nelhage.com/post/transparent-hugepages/>。

更多的内核参数信息可以参考 <https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt> 和 <https://www.kernel.org/doc/Documentation/sysctl/>。

21.4 索引层

21.4.1 使用全局模板

从 ES 5.x 开始，索引级别的配置需要写到模板中，而不是 `elasticsearch.yml` 配置文件，但是我们需要一些索引级别的全局设置信息，例如，`translog` 的刷盘方式等，因此我们可以将这些设置编写到一个模板中，并让这个模板匹配全部索引“*”，这个模板我们称为全局模板，例如：

```
{
  "template": "*",
  "order" : 0,
  "settings": {
    "index.number_of_replicas" : "1",
    "index.number_of_shards" : "24"
  }
}
```

`order` 为 0 代表该模板有最小的优先级。当索引创建时，ES 判断都匹配到哪些模板，如果匹配到多个模板，则将模板中的参数进行合并。当遇到冲突的设置项时，根据模板优先级 `order` 来决定谁的配置会生效。我们为全局模板设置最低的优先级，任何其他索引自定义的模板都可以覆盖它的设置。

21.4.2 索引轮转

如果有一个索引每天都有新增内容，那么不要让这个索引持续增大，建议使用日期等规则按一定频率生成索引。同时将索引设置写入模板，让模板匹配这一系列的索引，还可以为索引生成一个别名关联部分索引。我们一般按天生成索引，例如，要新增一个名为 `dns_log` 的索引，我们先创建模板，在模板中描述该索引的设置和 `mapping` 信息：

```
{
  "template": "dns_log-*",
  "settings": {
    "index.refresh_interval": "120s"
  },
  "mappings": {
    "dns_log": {
      "dynamic": false,
      "properties": {
        "field1": { "type": "text" },
        "field2": { "type": "integer", "doc_values": false }
      }
    }
  }
}
```

该模块会匹配 `dns_log-*` 规则的索引，对匹配规则的索引应用模板设置。写入程序每天生成一个索引，例如，`dns_log-20180614`。在该索引中只写入当天的数据。在搜索时，可以使用索引前缀 `dns_log-*` 进行搜索。当需要删除旧数据时，可以按日期删除索引的旧数据，删除索引会立即删除磁盘文件，释放存储空间。而如果不这么做，只删除部分 `doc`，则依赖 Lucene 分段的合并过程才能释放空间。

21.4.3 避免热索引分片不均

默认情况下，ES 的分片均衡策略是尽量保持各个节点分片数量大致相同。但是当集群扩容时，新加入集群的节点没有分片，此时新创建的索引分片会集中在新节点上，这导致新节点拥有太多热点数据，该节点可能会面临巨大的写入压力。因此，对于一个索引的全部分片，我们需要控制单个节点上存储的该索引的分片总数，使索引分片在节点上分布得更均匀一些。

例如，10 个节点的集群，索引主分片数为 5，副本数量为 1，那么平均下来每个节点应该有 $(5 \times 2) / 10 = 1$ 个分片，考虑到节点故障、分片迁移的情况，可以设置节点分片总数为 2：

```
curl -X PUT http://127.0.0.1:9200/myindex/_settings -d '{
  "index": { "routing.allocation.total_shards_per_node" : "2" }
}'
```

通常，我们会把 `index.routing.allocation.total_shards_per_node` 与索引主分片数、副本数等信息统一写到索引模板中。

21.4.4 副本数选择

由于搜索使用较好的硬件配置，硬件故障的概率相对较低。在大部分场景下，将副本数 `number_of_replicas` 设置为 1 即可。这样每个分片存在两个副本。如果对搜索请求的吞吐量要求较高，则可以适当增加副本数量，让搜索操作可以利用更多的节点。如果在项目初始阶段不知道多少副本数够用，则可以先设置为 1，后期再动态调整。对副本数的调整只会涉及数据复制和网络传输，不会重建索引，因此代价较小。

21.4.5 Force Merge

对冷索引执行 Force Merge 会有许多好处，我们在之前的章节中曾多次提到：

- 单一的分段比众多分段占用的磁盘空间更小一些；
- 可以大幅减少进程需要打开的文件 fd；
- 可以加快搜索过程，因为搜索需要检索全部分段；
- 单个分段加载到内存时也比多个分段更节省内存占用；
- 可以加快索引恢复速度。

可以选择在系统的空闲时间段对不再更新的只读索引执行 Force Merge：

```
curl -X POST "localhost:9200/twitter/_forcemerge"
```

该命令将分段合并为单个分段，执行成功后会自行 “flush”。

21.4.6 Shrink Index

需要密切注意集群分片总数，分片数越多集群压力越大。在创建索引时，为索引分配了较多的分片，但可能实际数据量并没有多大，例如，按日期轮询生成的索引，可能有些日子里数据量并不大，对这种索引可以执行 Shrink 操作来降低索引分片数量。Shrink 的例子可以参考 Shrink 分析一章。

我们可以为 Shrink Index 和上一节的 Force Merge 编写自动运行脚本，通过 crontab 选择在凌晨的某个时间对索引进行优化，编写 crontab 文件内容如下：

```
$cat escron
0 2 * * * es /home/es/software/elasticsearch/bin/index.sh
```


然后将定时任务添加到普通用户的定时任务中：

```
crontab escron
```

我们通常会为部署集群编写部署脚本，这些工作都可以放到部署脚本中。

21.4.7 close 索引

如果有些索引暂时不使用，则不会再有新增数据，也不会有对它的查询操作，但是可能以后会用而不能删除，那么可以把这些索引关闭，在需要时再打开。关闭的索引除存储空间外不占用其他资源。

通过下面的命令关闭或打开一个索引：

```
curl -X POST "localhost:9200/my_index/_close"
curl -X POST "localhost:9200/my_index/_open"
```

21.4.8 延迟分配分片

当一个节点由于某些原因离开集群时，默认情况下 ES 会重新确定主分片，并立即重新分配缺失的副分片。但是，一般来说节点离线是常态，可能因为网络问题、主机断电、进程退出等因素是我们经常面对节点离线的情况，而重新分配副分片的操作代价是很大的，该节点上存储的数据需要在集群上重新分配，复制这些数据需要大量带宽和时间，因此我们调整节点离线后分片重新分配的延迟时间：

```
"index.unassigned.node_left.delayed_timeout" : "5d"
```

这个索引级的设置写到模板的全局设置信息中，节点离线一般是暂时的，如果因为硬件故障，则修复时间一般是可以预期的，根据实际情况来调节这个延迟时间。

21.4.9 小心地使用 fielddata

聚合时，ES 通过 `doc_values` 获取字段值，但是 `text` 类型不支持 `doc_values`。当在 `text` 类型字段上聚合时，就会依赖 `fielddata` 数据结构，但 `fielddata` 默认关闭。因为它会消耗很多堆空间，并且在 `text` 类型字段上聚合通常没有什么意义。

`doc_values` 在索引文档时就会创建，而 `fielddata` 是在聚合、排序，或者脚本中根据需要动

态创建的。其读取每个分段中的整个倒排索引，反转 term 和 doc 的关系，将结果存储到 JVM 堆空间，这是非常昂贵的过程，会让用户感到明显的延迟。

fielddata 所占用的大小默认没有上限，可以通过 `indices.fielddata.cache.size` 来控制，该选项设置一个堆内存的百分比，超过这个百分后，使用 LRU 算法将老数据淘汰。

21.5 客户端

21.5.1 使用 REST API 而非 Java API

我们在第 1 章讨论过，由于 Java API 引起版本兼容性问题，以及微弱到可以忽略的性能提升，Java API 将在未来的版本中废弃，客户端最好选择 REST API 作为客户端，而不是 Java API。

21.5.2 注意 429 状态码

bulk 请求被放入 ES 的队列，当队列满时，新请求被拒绝，并给客户端返回 429 的状态码。客户端需要处理这个状态码，并在稍后重发请求。此刻客户端需要处理 bulk 请求中部分成功、部分失败的情况。这种情况产生在协调节点转发基于分片的请求到数据节点时，有可能因为对方的 bulk 队列满而拒绝写操作，而其他数据节点正常处理，于是客户端的 bulk 请求部分写入成功、部分写入失败。客户端需要将返回 429 的对应数据重试写入，而不是全部数据，否则写入的内容就会存在重复。

产生 429 错误是因为 ES 来不及处理，一般是由于写入端的并发过大导致的，建议适当降低写入并发。

21.5.3 curl 的 HEAD 请求

我们经常使用 curl 作为客户端进行一些日常操作。但是需要注意 curl 发送 HEAD 请求的方式并非我们预想的那样，例如，通过 HEAD 请求检查 doc 是否存在，官网的这个例子就是错误的：

```
curl -X HEAD "localhost:9200/twitter/_doc/0"
```

curl -X HEAD 只是将 HTTP 头部的方法设置为 HEAD，还会等待服务器返回 body，所以现象就是 curl 命令阻塞在那里。正确的方式应该是使用 -I 参数：

```
curl -I "localhost:9200/twitter/_doc/0"
```



使用 `-I` 参数 `curl` 会将 HTTP 方法设置为 `HEAD`，并在收到服务器返回的 HTTP 头部信息后关闭 TCP 连接。

21.5.4 了解你的搜索计划

就像在执行一条 SQL 语句时，需要了解其执行计划一样，我们需要知道一个搜索操作可能会命中多少分片，它执行的任务复杂性有多大，聚合范围有多大等情况。只有了解了搜索指令的执行代价，才能更好地使用 ES 进行搜索。例如，搜索应该只让尽量少的分片参与工作，如果只需要检索当天的内容，则在按天生成的索引中，只搜索当天的单个索引即可。通过日期范围查询会让其他天的索引不必要地执行一次搜索。

除了人为评估查询语句，还可以使用 `Profile API` 分析会命中哪些分片，每个分片执行的查询时间等细节。

21.5.5 为读写请求设置比较长的超时时间

读写操作都有可能是比较长的操作，例如，写一个比较大的 `bulk` 数据，或者执行较大范围的聚合。此时客户端为请求设置的超时时间应该尽量长，因为即使客户端断开连接，ES 仍然会在后台将请求处理完，如果超时设置比较短，则在密集的请求时会对 ES 造成非常大的压力。

21.6 读写

21.6.1 避免搜索操作返回巨大的结果集

我们在搜索流程中讨论过，由于协调节点的合并压力，所有的搜索系统都会限制返回的结果集大小，如果确实需要很大的结果集，则应该使用 `Scroll API`。

21.6.2 避免索引巨大的文档

`http.max_context_length` 的默认值为 100MB，ES 会拒绝索引超过此大小的文档，可以增加这个值，但 `Lucene` 仍然有大约 2GB 的限制。

即使不考虑这些限制，大型文档通常也不实用。大型文档给网络、内存和磁盘造成了更大的压力。即使搜索操作设置为不返回 `_source`，ES 总要获取 `_id`，对于大型文档来说，获取这个



字段的代价是很大的，这是由于操作系统的 cache 机制决定的。索引一个文档需要一些内存，所需内存大小是原始文档大小的几倍。邻近（Proximity）搜索（例如，短语查询）和高亮也会变得更加昂贵，因为它们的成本直接取决于原始文档大小。

因此可能要重新考虑信息的单位，例如，想要为一本书建立索引使之可以被搜索，这并不意味着把整本书的内容作为单个文档进行索引。最好使用章节或段落作为文档，然后在文档中加一个属性标识它们属于哪本书。这样不仅避免了大文档的问题，还使搜索的体验更好。

21.6.3 避免使用多个 `_type`

`_type` 本来是用于区分存储到同一个索引中的不同格式的数据，但是实际上面对这种情况应该用不同的索引解决，而不是在同一个索引中使用不同的 `_type`。因为不能通过 `_type` 来删除数据，这和通过 `_id` 删除数据没什么区别。而且还容易给初学者造成“index 就像数据库，`_type` 就像表”的误解。`_type` 是完全没必要存在的。

从 ES 6.0 开始，索引只允许存在一个 `_type`，7.0 版本之后将完全废弃了 `_type` 的概念。

21.6.4 避免使用 `_all` 字段

在写入速度优化中讨论过 `_all` 使用字段带来的负面影响，从 ES 6.0 开始，`_all` 字段默认被禁用，并且不建议使用。此类需求可以通过 mapping 中的 `copy_to` 参数创建自定义的 `_all` 字段。参考官方手册：<https://www.elastic.co/guide/en/elasticsearch/reference/master/copy-to.html>。

21.6.5 避免将请求发送到同一个协调节点

无论索引文档还是执行搜索请求，客户端都应该避免将请求发送到固定的某个或少数几个节点，因为少数几个协调节点作为整个集群对外的读写节点的情况下，它们很有可能承受不了那么多的客户端请求。尤其是搜索请求，协调节点的合并及排序会占用比较高的内存和 CPU，聚合会占用更多内存。因此会导致给客户端的返回慢，甚至导致节点 OOM。

正确的做法是将请求轮询发送到集群所有节点，如果使用 REST API，则可以在构建客户端的客户端对象时传入全部节点列表。如果在前端或脚本中访问 ES 集群，则可以部署 LVS，客户端使用虚 IP，或者部署 Nginx 使用反向代理。



21.7 控制相关度

通过 Painless 脚本控制搜索评分

ES 有多种方式控制对搜索结果的评分, 如果常规方式无法得到想要的评分结果, 则可以通过脚本的方式完全自己实现评分算法, 以得到预期的评分结果。ES 支持多种脚本语言, 经历各版本演变后, 从 5.0 版本开始实现了自己专用的语言: Painless。Groovy 脚本已弃用。Painless 是内置支持的, 脚本内容通过 REST 接口传递给 ES, ES 将其保存在集群状态中。在 5.x 版本中可以放到 config/scripts 下, 6.x 版本中只能通 REST 接口写入。

通过脚本控制评分的原理是编写一个自定义脚本, 该脚本返回评分值, 该分值与原分值进行加法等运算, 从而完全控制了评分算法。

例如, 我们有一个通讯录的名单索引 user_info, 为了简便说明问题, 索引只有一个 name 字段:

```
PUT user_info
{
  "mappings":{
    "user": {
      "properties": {
        "name":{
          "type":"keyword"
        }
      }
    }
  }
}
```

写入下列测试数据:

```
POST /user_info/_bulk
{"index" : {"_type":"user","_id":"1"}}
{"name":"高 X"}
{"index" : {"_type":"user","_id":"2"}}
{"name":"高 XX"}
{"index" : {"_type":"user","_id":"3"}}
```





```
{ "name": "X 高 X" }
{ "index" : { "_type": "user", "_id": "4" } }
{ "name": "X 高 X" }
{ "index" : { "_type": "user", "_id": "5" } }
{ "name": "XX 高" }
{ "index" : { "_type": "user", "_id": "6" } }
{ "name": "高 XXX" }
```

我们期望的返回顺序与两个原则有关：关键词出现的位置越靠前，排序应该越靠前；字段值约短，说明匹配度越高，排序应该越靠前。

因此，理想的顺序应该是：

```
高 X
高 XX
高 XXX
X 高 X
X 高 X
XX 高
```

下面执行搜索：

```
GET user_info/_search?size=20
{
  "query": {
    "query_string" : {
      "query" : "(name:(*高*))"
    }
  }
}
```

实际返回结果顺序如下，每一项的得分（`_score`）都是 1.0。

```
X 高 X
高 XX
X 高 X
高 XXX
高 X
XX 高
```





我们编写一个简单的脚本，通过 `doc['name'].value` 获取文档值，然后根据位置和相似度分别计算评分，将结果乘以不同权重再相加。

```
double position_score = 0;
double similarity_score = 0;

int pos = doc['name'].value.indexOf(params.keyword);
if (pos != -1)
{
    position_score = 10 - pos;
    if (position_score < 0) position_score = 0; //出现位置大于 10 的忽略其重要性
}

double similarity = Math.abs(1.0*doc['name'].value.length() -
params.keyword.length());
similarity_score = 10 - similarity;
if (similarity_score < 0) similarity_score = 0; //相似度差 10 个字符的忽略其重
//要性

//在下面调节各分值的权重
return position_score*0.6+similarity_score*0.4;
```

去掉脚本中的注释，并格式化为单行（在 sublime 中可以通过 Ctrl+A、Ctrl+J 组合键实现）内容后，写入 ES：

```
POST _scripts/user_info_score
{
  "script": {
    "lang": "painless",
    "source": "格式化为单行的脚本内容"
  }
}
```

我们的脚本 id 为 `user_info_score`，在 `script_score` 函数中指定脚本 id，再次执行查询：

```
GET user_info/_search?size=20
{
```





```
"query": {
  "function_score": {
    "query": {
      "query_string": {
        "query": "(name:(*高*))"
      }
    },
    "script_score": {
      "script": {
        "id": "user_info_score",
        "params": {
          "keyword": "高"
        }
      }
    },
    "boost_mode": "sum"
  }
}
```

`function_score` 查询是用来控制评分的终极武器，它允许为每个与主查询匹配的文档应用一个内置或自定义函数，以达到改变原始查询评分 `_score` 的目的。其中的 `script_score` 用于指定自定义脚本。`params` 指定作为变量传递到脚本中的参数。

`boost_mode` 字段用来指定新计算的分数与 `_score` 的结合方式，取值可以是：`multiply` 相乘（默认）、`replace` 替换、`_score sum` 相加、`avg` 取平均值、`max` 取最大值、`min` 取最小值。

这次查询返回了我们期望的结果：

```
"hits" : [
  {
    "_score" : 10.6,
    "_source" : {"name" : "高 X"}
  },
  {
    "_score" : 10.2,
    "_source" : {"name" : "高 XX"}
  },
  {
```





```
"_score" : 9.8,
"_source" : {"name" : "高 XXX"}
},
{
  "_score" : 9.6,
  "_source" : {"name" : "X 高 X"}
},
{
  "_score" : 9.6,
  "_source" : {"name" : "X 高 X"}
},
{
  "_score" : 9.0,
  "_source" : {"name" : "XX 高"}
}
]
```



22 chapter

第 22 章 故障诊断

当集群出现故障时，不必担心，我们有许多方法和工具来分析问题。在大部分情况下，我们遇到的问题都是由一些简单的原因导致的。但由于分布式系统的复杂性，有时候故障现象只出现了一次，并且难以复现，这就需要采取一些措施来缩小可疑的问题范围，虽然不能立刻解决问题，但是可以向前迈进一步。

综合来说，当遇到故障时，分析问题有两种思路：

(1) 从故障的具体现象和具体信息出发，逻辑性地向上推理可能的因素，并逐步排除，渐渐缩小问题范围，直到定位问题。这是系统化调试的思想，推荐阅读 Andreas Zeller 所著的《系统化调试指南》。

(2) 根据故障信息和经验直接猜测故障原因，这种凭空设想在面对简单问题时能比较快速定位，但在面对错综复杂、多种因素混合的问题时更多地需要理性推导。尽管如此，“假设故障原因”仍然很重要，在推导问题过程中，以及不可重现的问题时，需要联想到与其相关的都有哪些因素。

在逐步缩小问题范围、验证想法的过程中，验证方法可能有多种选择。最好选能证明问题，同时是最简单的方式。有些方法不一定能很严谨地验证问题，那这种验证方式的参考性和可信度就不大。同时有些方法可能操作复杂，有些可能耗时比较长，这就需要我们做出选择。

下面我们讨论一些通用的分析问题方法和工具。



22.1 使用 Profile API 定位慢查询

有时在发起一个查询时，查询会被延迟执行，或者响应时间很慢，查询缓慢可能会有多种原因：分片问题，或者计算查询中的某些元素。ES 从 2.2 版本开始提供 Profile API，供用户检查查询执行时间和其他详细信息。

Profile API 返回所有分片的详细信息。我们使用一个例子来演示其使用方式。为了简单起见，我们创建一个只有 1 个主分片、没有副分片的索引，然后写入若干文档：

```
curl -XPOST http://localhost:9200/myindex/mytype/1 -d '{
  "brand" : "Cotton Plus"
}'
curl -XPOST http://localhost:9200/myindex/mytype/2 -d '{
  "brand" : "Van Huesen"
}'
curl -XPOST http://localhost:9200/myindex/mytype/3 -d '{
  "brand" : "Arrow"
}'
```

使用 Profile API 来看一下检索的返回信息，可以通过在 query 部分上方提供 "profile": true 来启用 Profile API。

```
curl -XPOST http://localhost:9200/myindex/mytype/_search -d '{
  "profile": true,
  "query": {
    "match": {
      "brand": "Cotton Plus"
    }
  }
}'
```

Profile API 的结果是基于每个分片计算的。由于在我们的例子中只有一个分片，在 Profile API 响应的分片数组中只有一个数组元素，如下所示。

```
{
  "shards": [
    {
      "id": "[egPYczsCRTqaeJ8jKhFjtw][myindex][0]",
```



```
"searches": [  
  {  
    "query": [  
      {  
        "query_type": "BooleanQuery",  
        "lucene": "brand:levi brand:goals",  
        "time": "1.293761000ms",  
        "breakdown": {  
          "score": 0,  
          "create_weight": 136078,  
          "next_doc": 0,  
          "match": 0,  
          "build_scorer": 1082113,  
          "advance": 0  
        },  
      },  
      "children": [  
        {  
          "query_type": "TermQuery",  
          "lucene": "brand:levi",  
          "time": "0.04626300000ms",  
          "breakdown": {  
            "score": 0,  
            "create_weight": 30190,  
            "next_doc": 0,  
            "match": 0,  
            "build_scorer": 16073,  
            "advance": 0  
          }  
        },  
        {  
          "query_type": "TermQuery",  
          "lucene": "brand:goals",  
          "time": "0.02930700000ms",  
          "breakdown": {  
            "score": 0,  
            "create_weight": 16600,  
            "next_doc": 0,  
            "match": 0,  
          }  
        }  
      ]  
    }  
  ]  
}
```

```

        "build_scorer": 12707,
        "advance": 0
      }
    }
  ]
},
"rewrite_time": 64032,
"collector": [
  {
    "name": "TotalHitCountCollector",
    "reason": "search_count",
    "time": "0.002931000000ms"
  }
]
}
]
}
}

```

上面的响应显示的是单个分片。每个分片都被分配一个唯一的 ID，ID 的格式是 `nodeID[shardID]`。现在在 “shards” 数组里还有另外三个元素，它们是：

- (1) query。
- (2) rewrite_time。
- (3) collector。

下面解释每个元素的意义。

1. Query

Query 段由构成 Query 的元素及它们的时间信息组成。Profile API 结果中 Query 部分的基本组成如下：

- `query_type`，它向我们显示了哪种类型的查询被触发。此处是布尔值。因为多个关键字匹配查询，因此被分成两个布尔查询。
- `lucene`，该字段显示启动查询的 Lucene 方法。这里是 “brand:levi brand:goals”。
- `time`，Lucene 执行此查询所用的时间。单位是毫秒。
- `breakdown`，有关查询的更详细的细节，主要与 Lucene 参数有关。

- `children`，具有多个关键字的查询被拆分成相应术语的布尔查询，每个查询都作为单独的查询来执行。每个子查询的详细信息将填充到 `Profile API` 输出的子段中。可以看到第一个子元素查询是“`levi`”，下面给出查询时间和其他 `breakdown` 参数等详细信息。同样，对于第二个关键字，有一个名为“`goals`”的子元素具有与其兄弟相同的信息。从查询中的子段中，我们可以得到关于哪个搜索项在总体搜索中造成最大延迟的信息。

2. Rewrite Time

由于多个关键字会分解以创建个别查询，所以在这个过程中肯定会花费一些时间。将查询重写一个或多个组合查询的时间被称为“重写时间”（单位为纳秒）。

3. Collectors

在 `Lucene` 中，收集器负责收集原始结果，并对它们进行组合、过滤、排序等处理。例如，在上面的执行的查询中，当查询语句中给出 `size:0` 时，使用的收集器是“`totalHitCountCollector`”。这只返回搜索结果的数量（`search_count`），不返回文档。此外，收集者所用的时间也一起给出了。

`Profile API` 非常有用，它让我们清楚地看到查询时间。通过向我们提供有关子查询的详细信息，我们可以清楚地知道在哪个环节查询慢，这是非常有用的。另外，在 `API` 返回的结果中，关于 `Lucene` 的详细信息也让我们深入了解到 `ES` 是如何执行查询的。

22.2 使用 Explain API 分析未分配的分片(Unassigned Shards)

一个 `ES` 索引由多个分片组成，由于某些原因，某些分片可能会处于未分配状态（`Unassigned`），导致集群健康处于 `Yellow` 或 `Red` 状态，这是一种比较常见的错误信息，导致分片处于未分配的原因可能是节点离线、分片数量设置错误等原因，使用 `Explain API` 可以很容易分析当前的分片分配情况。这个 `API` 主要为了解决下面两个问题：

- （1）对于未分配的分片，给出为什么没有分配的具体原因。
- （2）对于已分配的分片，给出为什么将分片分配给特定节点的理由。

接下来我们通过几个例子来演示如何通过 `Explain API` 来定位分片分配问题。

22.2.1 诊断未分配的主分片

我们创建一个名为 `test_idx` 的索引，该索引只有一个主分片，没有副分片。集群有两个节点，名为 `A` 和 `B`。但是在创建索引时，我们设置分配过滤器，使其不能被分配到节点 `A` 和 `B` 上：

```
PUT /test_idx?wait_for_active_shards=0
{
  "settings":
  {
    "number_of_shards": 1,
    "number_of_replicas": 0,
    "index.routing.allocation.exclude._name": "A,B"
  }
}
```

虽然索引能创建成功，但是因为过滤规则的限制，索引分片无法分配到集群仅有的 A 和 B 两个节点。此时集群处于 Red 状态，我们通过 Explain API 来获取第一个未分配分片的原因解释（在本例中，集群中只有一个未分配的分片）。

```
GET /_cluster/allocation/explain
```

返回信息摘要如下：

```
{
  "index" : "test_idx",
  "shard" : 0,
  "primary" : true,
  "current_state" : "unassigned",
  "unassigned_info" : {
    "reason" : "INDEX_CREATED",
    "last_allocation_status" : "no"
  },
  "can_allocate" : "no",
  "allocate_explanation" : "cannot allocate because allocation is not
permitted to any of the nodes",
  "node_allocation_decisions" : [
    {
      "node_id" : "tn3qdPdnQWuumLxVVjJJYQ",
      "node_name" : "A",
      "node_decision" : "no",
      "weight_ranking" : 1,
      "deciders" : [
        {
```



```

        "decider" : "filter",
        "decision" : "NO",
        "explanation" : "node matches index setting [index.routing.
allocation.exclude.] filters [_name:\"A OR B\"]"
    }
}
},
{
    "node_id" : "qNgMCvaCSPi3th0mTcyvKQ",
    "node_name" : "B",
    "node_decision" : "no",
    "weight_ranking" : 2,
    "deciders" : [
        {
            "decider" : "filter",
            "decision" : "NO",
            "explanation" : "node matches index setting [index.routing.
allocation.exclude.] filters [_name:\"A OR B\"]"
        }
    ]
}
]
}
}

```

Explain API 给出了该分片未分配的原因，由于索引刚创建（unassigned_info），它处于未分配状态（current_state）。由于没有节点允许分配给该分片（allocate_explain），所以无法分配分片（can_allocation）。深入每个节点的决策信息（node_allocation_decisions），可以看到由于索引的过滤器设置，分配操作被 decider 拦截。decider 给出了具体的 decider 名称，接下来是决策结果及具体的原因（explanation）。

下面更新分配过滤器设置：

```

PUT /test_idx/_settings
{
    "index.routing.allocation.exclude._name": null
}

```

然后重新运行 Explain API，将收到以下结果：

```
unable to find any unassigned shards to explain
```

表示当前没有未分配的分片。我们还可以在主分片上运行 Explain API：

```
GET /_cluster/allocation/explain
{
  "index": "test_idx",
  "shard": 0,
  "primary": true
}
```

我们可以看到分片被分配到了哪个节点，返回信息摘要如下：

```
{
  "index" : "test_idx",
  "shard" : 0,
  "primary" : true,
  "current_state" : "started",
  "current_node" : {
    "id" : "tn3qdPdnQWuumLxVVjJJYQ",
    "name" : "A",
    "weight_ranking" : 1
  },
  ...
}
```

可以看出分片被成功（started）分配到节点 A。

现在，我们向索引 test_idx 中写入一些数据，然后停掉节点 A，由于该索引没有副分片，所以集群变为 Red 状态。在主分片上重新运行 Explain API，返回信息摘要如下：

```
{
  "index" : "test_idx",
  "shard" : 0,
  "primary" : true,
  "current_state" : "unassigned",
  "unassigned_info" : {
    "reason" : "NODE_LEFT",
```

```

    "details" : "node_left[qU98BvbtQu2crqXF2ATFdA]",
    "last_allocation_status" : "no_valid_shard_copy"
  },
  "can_allocate" : "no_valid_shard_copy",
  "allocate_explanation" : "cannot allocate because a previous copy of the primary
    shard existed but can no longer be found on the nodes in the cluster"
}

```

输出信息告诉我们主分片处于未分配状态（`current_state`），原因是持有此分片的节点离线（`unassigned_info`）。`Explain API` 给出了分片不能分配的原因，是因为集群中没有任何分片 0 的有效副本（`can_allocate`）。`allocate_explanation` 字段给出了详细的解释。

`Explain API` 告诉我们主分片不再有效的分片副本，我们知道持有该分片的节点离线了，此时唯一的办法就是等待节点重新加入集群。在极端情况下，节点永久移除，只能接受丢失数据的现实，并通过 `reroute API` 重新分配空的主分片。

22.2.2 诊断未分配的副分片

我们把刚才的索引 `test_idx` 副本数调整为 1:

```

PUT /test_idx/_settings
{
  "number_of_replicas": 1
}

```

索引 `test_idx` 现在有两个分片，分片 0 的主分片及分片 0 的副分片。因为 A 节点已经存储了主分片，所以副分片将被分配到节点 B，以达到集群均衡的目的。现在在副分片上运行 `Explain API`:

```

GET /_cluster/allocation/explain
{
  "index": "test_idx",
  "shard": 0,
  "primary": false
}

```

返回信息摘要如下:

```

{
  "index" : "test_idx",

```

```

    "shard" : 0,
    "primary" : false,
    "current_state" : "started",
    "current_node" : {
      "id" : "qNgMCvaCSPi3th0mTcyvKQ",
      "name" : "B",
      "weight_ranking" : 1
    },
    ...
  }

```

输出信息显示分片已经被分配到节点 B，并且为 started 状态。

接下来，我们再次设置索引的分配过滤器，但是这次阻止分配分片的节点 B：

```

PUT /test_idx/_settings
{
  "index.routing.allocation.exclude._name": "B"
}

```

现在重启节点 B，在副分片上重新运行 Explain API，返回信息摘要如下：

```

{
  "index" : "test_idx",
  "shard" : 0,
  "primary" : false,
  "current_state" : "unassigned",
  "unassigned_info" : {
    "reason" : "NODE_LEFT",
    "details" : "node_left[qNgMCvaCSPi3th0mTcyvKQ]",
    "last_allocation_status" : "no_attempt"
  },
  "can_allocate" : "no",
  "allocate_explanation" : "cannot allocate because allocation is not
permitted to any of the nodes",
  "node_allocation_decisions" : [
    {
      "node_name" : "B",
      "node_decision" : "no",

```

```

    "deciders" : [
      {
        "decider" : "filter",
        "decision" : "NO",
        "explanation" : "node matches index setting [index.routing.
allocation.exclude.] filters [_name:\"B\"]"
      }
    ],
    {
      "node_name" : "A",
      "node_decision" : "no",
      "deciders" : [
        {
          "decider" : "same_shard",
          "decision" : "NO",
          "explanation" : "the shard cannot be allocated to the same node on
which a copy of the shard already
exists [[test_idx][0], node[tn3qdPdnQWuumLxVVjJJYQ], [P], s[STARTED],
a[id=JNODiTgYTrSp8N2s0Q7MrQ]]"
        }
      ]
    }
  ]
}

```

结果显示副分片当前处于未分配状态（`can_allocate`），因为分配过滤设置了禁止把分片分配到节点 B 上（`explanation`）。因为节点 A 上已经指派了主分片，所以不允许再把该分片的其他副本分配到 A 节点（`explanation`）。ES 会避免将主副分片分配到同一个节点，主要是为了防止当节点失效时所有副本都不可用，以及可能的数据丢失。

22.2.3 诊断已分配的分片

如果分片可以正常分配，为什么还要关注 `explain` 信息呢？一个常见的原因是想将分片手工迁移到某个节点，但是出于某些原因分片没有迁移，这时 `Explain API` 帮助我们展示其中原因。

我们清除 `test_idx` 索引的过滤器设置，使主副分片都可以正常分配。


```
PUT /test_idx/_settings
{
  "index.routing.allocation.exclude._name": null
}
```

然后重新设置分配过滤器，使主分片从当前节点迁移走：

```
PUT /test_idx/_settings
{
  "index.routing.allocation.exclude._name": "A"
}
```

我们期望的结果是主分片从节点 A 移动到另一个节点，然而事与愿违，在主分片上运行 Explain API，看看具体原因：

```
GET /_cluster/allocation/explain
{
  "index": "test_idx",
  "shard": 0,
  "primary": true
}
```

返回信息摘要如下：

```
{
  "index" : "test_idx",
  "shard" : 0,
  "primary" : true,
  "current_state" : "started",
  "current_node" : {
    "name" : "A",
  },
  "can_remain_on_current_node" : "no",
  "can_remain_decisions" : [
    {
      "decider" : "filter",
      "decision" : "NO",
      "explanation" : "node matches index setting [index.routing.allocation.exclude.] filters [_name:\"A\"]"
    }
  ]
}
```

```

    }
  ],
  "can_move_to_other_node" : "no",
  "move_explanation" : "cannot move shard to another node, even though it
is not allowed to remain on its current node",
  "node_allocation_decisions" : [
    {
      "node_name" : "B",
      "node_decision" : "no",
      "weight_ranking" : 1,
      "deciders" : [
        {
          "decider" : "same_shard",
          "decision" : "NO",
          "explanation" : "the shard cannot be allocated to the same node on
which a copy of the shard already
exists [[test_idx][0], node[qNgMCvaCSPi3th0mTcyvKQ], [R], s[STARTED],
a[id=dNgHLTKwRH-Dp-rIX4Hkqg]]"
        }
      ]
    }
  ]
}

```

从返回结果可以看出，主分片仍然被分配给节点 A（`current_node`），集群知道这个分片不应该保留在当前节点（`can_remain_on_current_node`），原因是主分片被当前节点的 `decider` 拦截（`can_remain_decisions`）。尽管分片不允许保留在当前节点，但它也不能移动到其他节点（`can_move_to_other_node`），因为被节点 B 的 `decider` 拦截，主副分片不能分配到同一节点。

`Explain API` 是诊断生产环境分片分配过程的利器，它为定位问题节省了很多时间。关于该命令的完整参数请参考官方手册：<https://www.elastic.co/guide/en/elasticsearch/reference/current/search-explain.html>。

22.3 节点 CPU 使用率高

节点占用 CPU 很高，我们想知道 CPU 在运行什么任务，一般通过线程堆栈来查看。有两种方式可以查看哪些线程 CPU 占用率比较高。

- 使用 `hot_threads API`，参考 `ThreadPool` 一章。

- 使用 top+jstack, top 获取线程级 CPU 占用率, 再根据线程 ID, 配合 jstack 定位 CPU 占用率在特定比例之上的线程堆栈。

推荐使用 hot_threads API 获取繁忙线程的堆栈。top+jstack 的方式由于两个命令的执行在时间上（对进程采样的时间点）存在误差, 所以定位出的堆栈存在一定概率的偏差。为了降低这种误差, 可以在脚本中让两个命令同时执行, 参考下面的实现:

```
#!/bin/bash

pid=$1
cpu_use=$2

if [ $# -lt 1 ] ; then
    echo "USAGE: $0 pid [cpu_use]"
    echo " e.g.: $0 23979"
    exit 1;
fi

if test -z "$cpu_use" ;then
    cpu_use=20
fi

grep_jstack()
{
    is_start="false"
    while read line
    do
        if echo "nid=0x"${line}|grep -q $1 ; then
            is_start="true"
            elif [[ "$is_start" = "true" ]] && [ -z "${line}" ]; then
                break;
            fi

            if [ "$is_start" = "true" ] ;then
                echo ${line}
            fi
        done < /tmp/s-$pid
    }
```

```
grep_all()
{
    while read line
    do
        if [ -n "${line}" ]; then
            grep_jstack ${line}
            echo "-----"
        fi
    done </tmp/j-$pid
}

jstack $pid > /tmp/s-$pid &
jstack_pid=$!

top -H -b -n 1 -p $pid | sed -n '8,$p' | awk -v val=$cpu_use ' $9>val {printf("%x\n",
$1);fflush()}' > /tmp/j-$pid

wait $jstack_pid
grep_all

echo "-----THE END-----\n\n"
```

22.4 节点内存使用率高

节点内存使用率高，我们想要知道内存是被哪些数据结构占据的，通用方式是用 jmap 导一个堆出来，加载到 MAT 中进行分析，可以精确定位数据结构占用内存的大小，以及被哪些对象引用。这是定位此类问题最直接的方式。

jmap 导出的堆可能非常大，操作比较花时间，我们也可以简单看一下 ES 中几个占用内存比较大的数据结构，有些数据结构无法看到其当前的实际大小，只能通过设置的上限粗略评估。

bulk 队列 可以通过_cat API 查看 bulk 队列中当前的使用量，任务总数乘以 bluk 请求的大小就是占用内存的大小。如果队列设置很大，则在写入压力大的时候就会导致比较高的内存占用。默认值为 200，一般情况下都够用了。

Netty 缓冲 在一些特别的情况下，Netty 的内存池也可能会占用比较高的内存。Netty 收到一个客户端请求时，为连接分配内存池，客户端发送的数据存储到 Netty 的内存池中，直到 ES

层处理完上层逻辑，回复客户端时，才释放该内存。当 ES 收到客户端请求时，如果在处理完毕之前客户端关闭连接，则 ES 依然会把这个请求处理完，只是最后才出现回复客户端失败。这个过程可能会导致内存累积，例如，执行 bulk 请求时，客户端发送完毕，不等 ES 返回响应就关闭连接，然后立即发起下一个请求，结果这些请求实际上都在等待处理，就可能占用非常多的内存。所以客户端的请求超时时间应该尽量设置得长一些，建议设置为分钟级。

indexing buffer 索引写入缓冲用于存储索引好的文档数据，当缓冲满时，生成一个新的 Lucene 分段。在一个节点上，该节点的全部分片共享 indexing buffer。该缓冲默认大小为堆内存的 10%，加大该缓冲需要考虑到对 GC 的压力。

超大数据集的聚合 协调节点对检索结果进行汇总和聚合，当聚合涉及的数据量很大时，协调节点需要拉取非常多的内容，大范围的聚合是导致节点 OOM 的常见原因之一。

分段内存 一个 Lucene 分段就是一个完整的倒排索引，倒排索引由单词词典和倒排列表组成。在 Lucene 中，单词词典中的 FST 结构会被加载到内存。因此每个分段都会占用一定的内存空间。可以通过下面的 API 来查看某个节点上的所有分段占用的内存总量：

```
curl -X GET "localhost:9200/_cat/nodes?v&h=segments.memory"
```

```
6.7kb
```

也可以单独查看每个分段的内存占用量：

```
curl -X GET "localhost:9200/_cat/segments?v&h=index,shard,segment,
size.memory"
```

index	shard	segment	size.memory
website	1	_0	1195
website	1	_1	1187
website	1	_2	994

size.memory 字段代表分段的内存占用量。该 API 可以根据索引名称等进行过滤，完整的使用手册可以参考：<https://www.elastic.co/guide/en/elasticsearch/reference/master/cat-segments.html>。

Fielddata cache 在 text 类型字段上进行聚合和排序时会用到 Fielddata，默认是关闭的，如果开启了 Fielddata，则其大小默认没有上限，可以通过 indices.fielddata.cache.size 设置一个百分比来控制其使用的堆内存上限。可以通过下面的命令查看节点上的 Fielddata 使用情况：

```
curl -X GET "localhost:9200/_cat/nodes?v&h=fielddata.memory_size"
```

```
curl -XGET 'http://localhost:9200/_nodes/stats/indices/fielddata?fields=
field1,field2&pretty'
```


也可以查看索引级的 `Fielddata` 使用情况：

```
curl -XGET 'http://localhost:9200/_stats/fielddata/?fields=field1,field2&pretty'
```

完整的 `Nodes Stats API` 使用方式可以参考官方手册：<https://www.elastic.co/guide/en/elasticsearch/reference/master/cluster-nodes-stats.html>。

`cat nodes API` 可以参考：<https://www.elastic.co/guide/en/elasticsearch/reference/master/cat-nodes.html>。

Shard request cache 分片级别的请求缓存。每个分片独立地缓存查询结果。该缓冲默认是开启的，默认为堆大小的 1%，可以通过 `indices.requests.cache.size` 选项来调整。其使用 LRU 淘汰策略。默认情况下，只会缓存 `size=0`（结果为空）的请求，它并不缓存命中结果（hits），但是会缓存 `hits.total`、`aggregations` 和 `suggestions`。

是否开启该缓冲可以动态调整：

```
PUT /my_index/_settings
{ "index.requests.cache.enable": true }
```

也可以在某个请求中指定不使用缓存：

```
GET /my_index/_search?request_cache=false
```

可以使用下面的 API 来获取缓存使用量：

```
curl -X GET "localhost:9200/_stats/request_cache?pretty"
curl -X GET "localhost:9200/_nodes/stats/indices/request_cache?pretty"
curl -X GET "localhost:9200/_cat/nodes?v&h=request_cache.memory_size"
```

Node Query Cache 节点查询缓存由节点上的所有分片共享，也是一个 LRU 缓存，用于缓存查询结果，只缓存在过滤器上下文中使用的查询。该缓存默认开启，大小为堆大小的 10%。可以通过 `indices.queries.cache.size` 选项来配置大小，同时可以通过 `index.queries.cache.enabled` 选项在索引级启用或禁用该缓存。

该缓存的使用量可以通过下面的命令来获取：

```
curl -X GET "localhost:9200/_cat/nodes?v&h=query_cache.memory_size"
```

此外，ES 进程的内存使用量还与 Lucene 以 `mmap` 方式加载段文件相关。`mmap` 加载的文

件会被分配进程地址空间，因此它们同样算作 ES 占用的内存，我们可以通过 `pmap` 命令查看进程都有哪些文件被映射进来。通过 `mmap` 系统调用映射进来的段文件数据量通常都比较大，如果 `mmap` 带来的难以控制的内存占用对系统来说是个麻烦，则可以考虑调整存储类型：

```
index.store.type: niofs
```

将默认的 `mmapfs` 修改为 `niofs` 等类型，虽然使用 `mmap` 可以少一次内存拷贝，但是由于目前 Lucene 使用 `mmap` 时不控制它的预读方式，`mmap` 会预读取 2MB 的数据，在随机 I/O 的场景中，其效率未必会高于 NIO。具体可以参考 <https://github.com/elastic/elasticsearch/issues/27748>。

22.5 Slow Logs

当遇到查询慢的时候，想知道对方的查询语句是什么，在日志中记录所有查询语句可能会导致日志量太大，因此 ES 允许只将执行慢的请求记录日志，“慢”的程度可以自己定义。写入慢或查询慢的原因可能会有多种因素，慢日志对于我们诊断这些问题非常有用。

目前，ES 记录了两种类型的慢日志。

慢搜索日志 用来记录哪些查询比较慢，“慢”的程度由程序自己定义，每个节点可以设置不同的阈值。ES 的搜索由两个阶段组成：查询和取回。慢搜索日志给出了每个阶段所花费的时间，以及整个查询内容本身。

慢搜索日志有以下索引级配置项：

```
index.search.slowlog.threshold.query.warn
index.search.slowlog.threshold.query.info
index.search.slowlog.threshold.query.debug
index.search.slowlog.threshold.query.trace
index.search.slowlog.threshold.fetch.warn
index.search.slowlog.threshold.fetch.info
index.search.slowlog.threshold.fetch.debug
index.search.slowlog.threshold.fetch.trace
index.search.slowlog.level
```

慢搜索日志可以为查询和取回阶段单独设置以时间为单位的阈值，如果设置为 0，则输出全部搜索日志。在定义好每个级别的时间后，通过 `level` 决定输出哪个级别的日志。

日志输出内容样例如下：

```
[2017-09-10T12:35:53,352][WARN ][index.search.slowlog.query] [GOgO9TD]
```



```
[testindex-slowlogs][1] took[197.6micros], took_millis[0], types[], stats[],
search_type[QUERY_THEN_FETCH], total_shards[5], source[{"query":{"match":{"name":
{"query":"Nariko","operator":"OR","prefix_length":0,"max_expansions":50,"fuz
zy_transpositions":true,"lenient":false,"zero_terms_query":"NONE","boost":1.
0}}},"sort":[{"price":{"order":"desc"}}]}],
```

慢索引日志 用来记录哪些索引操作比较慢，其记录了哪些索引操作耗时比较长，阈值同样可以设置。与慢搜索不同，索引内容可能非常大，因此默认记录源文档内容的前 1000 行。可以设置为捕获整个文档，或者不记录原始文档本身。

慢索引日志有以下索引级配置项：

```
index.indexing.slowlog.threshold.index.warn
index.indexing.slowlog.threshold.index.info
index.indexing.slowlog.threshold.index.debug
index.indexing.slowlog.threshold.index.trace
index.indexing.slowlog.level
index.indexing.slowlog.source
```

与慢搜索日志的配置类似，同样可以定义 4 种不同的慢日志级别，为每个级别设置不同的时间，然后通过 level 来决定输出哪个级别的日志。如果阈值设置为 0，则将输出全部索引日志。source 参数设置了日志中捕获源文档的行数。

日志输出内容样例如下：

```
[2017-09-10T12:07:26,683][WARN ][index.indexing.slowlog.index] [GOgO9TD]
[testindex-slowlogs/yNbyYklARSW_hd0YRh6J0A] took[142.3micros], took_millis[0],
type[product], id[105], routing[] , source[{"price":9925,"name":"Nariko"}]
```

22.6 分析工具

本节介绍一些基础的分析工具，它们大部分是系统自带且常用的，无论分析性能问题还是排查故障，都离不开它们。

22.6.1 I/O 信息

iostat iostat 是用来分析 I/O 状态的常用工具，其输出结果是以 `/proc/diskstats` 为基础计算的。例如，我们使用 1 秒的间隔来采样：



```
iostat -xd 1
```

输出结果包括系统中全部磁盘的信息：

Device:	rrqm/s	wrqm/s	r/s	w/s	rsec/s	wsec/s	avgrq-sz	avgqu-sz	await	svctm	%util
sda	0.12	203.18	4.15	12.72	1170.51	1722.94	171.55	0.07	4.26	4.16	7.01
sdb	0.04	170.13	3.38	3.17	1387.51	1384.84	423.44	0.07	10.42	4.48	2.94

我们经常关注的几个指标：

- iops，由 r/s（每秒读次数）和 w/s（每秒写次数）组成。
- await，平均 I/O 等待时间，包括硬件处理 I/O 的时间和在队列中的等待时间。
- %util，设备的繁忙比，是设备执行的 I/O 时间与所经过的时间百分比。当值接近 100% 时设备产生饱和。在设备具有并行处理能力的情况下，util 达到 100% 不代表设备没有余力处理更多 I/O 请求。

当 I/O 产生性能问题时，iostat 可能不足以定位故障，可以使用 blktrace 来分析 I/O 请求的各个环节。该工具的原理和使用方式可以参考 <http://bean-li.github.io/blktrace-to-report/>。

进程级 I/O 状态 iostat 提供磁盘级的 I/O 状态，无法关联到进程，如果想查看哪些进程的 I/O 最高，则可以使用 pidstat 或 iotop 两个工具，它们可以动态给出每个进程的读写速度。例如，下面为 iotop 的输出结果，在统计信息中给出了全部磁盘的读写速度，以及每个进程的读写速度。

Total DISK READ: 0.00 B/s Total DISK WRITE: 23.69 K/s							
TID	PRI	USER	DISK READ	DISK WRITE	SWAPIN	IO>	COMMAND
19582	be/3	root	0.00 B/s	40.61 K/s	0.00 %	7.63 %	[jbd2/sda2-8]
11129	be/4	zookeepe	0.00 B/s	3.38 K/s	0.00 %	1.18 %	java -Dzookeeper.log.dir=/
28044	be/4	kafka	0.00 B/s	6.77 K/s	0.00 %	0.00 %	java -Xmx5G -Xms5G -server
27948	be/4	kafka	0.00 B/s	13.54 K/s	0.00 %	0.00 %	java -Xmx5G -Xms5G -server

如果想看到特定磁盘上特定进程的 I/O 情况，则这两个工具无法做到，这种情况可以通过 systemtap 来监控。

22.6.2 内存

top、free、vmstat 等工具可以帮助我们看到基础的内存信息，包括物理内存总量、剩余空间、cache 量等。这些简单指令本书不再一一列出。我们感兴趣的是，当系统物理内存不足时，系统回收内存的效率如何。在这种场景下，我们可以通过 sar -B 来观察内存分页的统计信息：

```
sar -B
```





```

09:30:01 AM    pgpgin/s    pgpgout/s      fault/s    majflt/s    pgfree/s    pgscank/s    pgscand/s
pgsteal/s    %vmeff
09:40:01 AM      74.39        3.27    210.37      0.51    952.68      0.00      5.53      4.86    87.95
09:50:01 AM      71.64        1.31    192.98      1.88    173.83     15.22      0.00     14.49    95.15
10:00:01 AM      20.07        2.04    489.02      0.18    361.66      6.39      3.48      9.37    95.00
10:10:02 AM    2534.52     12.42   1103.69      1.38    970.42     681.02      2.72    659.23    96.41
10:20:01 AM     131.43     13.93    383.28      0.34    434.35     17.93      0.00     14.93    83.28
Average:      566.67      6.59    475.95      0.86    578.65    144.19      2.35    140.65    95.98

```

输出结果中几个字段的含义如下。

- **pgfree/s**: 每秒被放入空闲列表中的页数，如果其他进程需要内存，则这些页可以被分页（paged out）。
- **pgscank/s**: 每秒被 kswapd 守护进程扫描的页数。
- **pgscand/s**: 每秒被直接扫描的页数。
- **pgsteal/s**: 为了满足内存需求，系统每秒从缓存（pagecache 和 swapcache）回收的页面数。
- **%vmeff**: 代表页面回收效率：计算方式为 $\text{pgsteal}/(\text{pgscand} + \text{pgscank})$ 。过低表明虚拟内存存在问题，如果在采样周期内没有发生页面扫描，则该值为 0 或接近 100。

当一个进程需要更多内存而实际空间不足时，就会发生页面扫描。内核检查页面，找出哪些页面需要分页（paged out）。独立地观察 pgscand、pgsteal 等数值一般没太多参考意义，除非异常高。通常我们可以重点关注 vmeff 的百分比，当值为 0 或接近 100 时代表内存够用，当值比较低时，（例如，低于 30%，甚至低于 1%）就需要小心，这时页面回收效率就存在问题。页面回收效率问题可以检查一下内核参数：vm.zone_reclaim_mode。CentOS 7 中其默认值为 0，当为 1 的时候，可能会导致回收效率低下。我们建议确保此值为 0，详细内容可参阅 NUMA 资料。

关于内存分页的更多信息可以参考维基 Paging: <https://en.wikipedia.org/wiki/Paging>。

另外一种情况，在开启了交换分区的系统上，可以通过 sar -W 查看页面交换情况：

```

sar -W

09:30:01 AM    pswpin/s    pswpout/s
09:40:01 AM      3.32      0.00
09:50:01 AM     13.18      0.00
10:00:01 AM      0.54      0.00
10:10:02 AM      0.96      1.37

```





```
10:20:01 AM      0.24      0.00
10:30:01 AM      0.15      0.00
10:40:01 AM      0.85      0.00
10:50:01 AM      0.16      0.00
Average:         2.43      0.17
```

- pswpin/s: 每秒系统换入交换页面（swap page）数量。
- pswpout/s: 每秒系统换出交换页面（swap page）数量。

发生页面交换会导致服务器性能严重下降，我们应该在生产环境关闭交换分区。

22.6.3 CPU 信息

基本信息 `vmstat` 输出用户级（us）和内核级（sy）的 CPU 占用百分比，以及采样周期内的上下文切换次数，block in、block out 次数等信息如下：

```
vmstat
```

```
procs -----memory----- --swap--  ---io---  -system--  -----cpu-----
r  b  swpd   free   buff   cache   si   so    bi    bo    in   cs us sy id wa st
3  0 1359516 86672   12 399056   56 227   645   235  405  241  2  1 97  0  0
```

`mpstat` 除了获取用户级（usr）和内核级（sys）的 CPU 占用百分比，还可以输出采样周期内的软中断（soft）、硬中断（irq）占用时间的百分比：

```
mpstat
```

```
09:19:11 PM CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest
%gnice  %idle
09:19:11 PM all    2.09    0.01    0.88    0.06    0.00    0.04    0.00    0.00
0.00   96.92
```

诊断导致 CPU 高的系统调用 正常情况下应用程序占用用户态 CPU 时间，如果进程占用 sys 比较高，则表示程序执行在内核态的操作非常耗费 CPU，我们可以使用一些工具来检查应用程序产生系统调用的统计信息。

`strace` `strace` 的 `-c` 参数可以统计系统调用次数和执行时间。例如：

```
strace -o output.txt -f -c -e trace=all -p 5233
```





`strace` 命令可以指定跟踪哪些类型的系统调用，例如，文件级、进程相关、网络相关等，也可以跟踪特定的系统调用，例如，`open`、`close`。我们的示例中选择跟踪所有调用。命令执行一段时间后需要手工按“`Ctrl+C`”组合键退出，获得以下输出结果：

% time	seconds	usecs/call	calls	errors	syscall
83.32	6.670477	381	17490	2829	futex
13.64	1.091976	3467	315		epoll_wait
2.00	0.160510	26752	6	3	restart_syscall
0.54	0.042992	10	4197		read
0.17	0.013824	4	3922		lseek

这个例子中显示主要的系统调用是 `futex`，占了系统调用的 83.32%，花费时间为 6.6 秒。

Perf

`perf` 是 Linux 用户主要的性能分析工具，它可以做的事情很多，现在我们用它分析 CPU。`perf top` 用来实时显示系统最耗时的内核函数及进程，动态输出 TopN 个结果，如下图所示。

Samples: 355 of event 'cpu-clock', Event count (approx.): 88750000		
Overhead	Shared Object	Symbol
18.31%	[kernel]	[k] module_get_kallsym
5.63%	[kernel]	[k] kallsyms_expand_symbol.constprop.1
5.35%	[kernel]	[k] vsnprintf
5.35%	perf	[.] hex2u64

我们还可以使用 `perf record` 来记录函数级别的统计信息，下列命令对系统采样 10 秒，并将采样数据输出到 `cycle.perf` 文件中。

```
sudo perf record -a -e cycles -o cycle.perf -g sleep 10
```

接下来，根据采样数据生成报告：

```
perf report -i cycle.perf | more
```

输出信息如下：





```
# Samples: 25K of event 'cpu-clock'
# Event count (approx.): 6467250000
#
# Children      Self  Command      Shared Object      Symbol
# .....
#
# 87.89%      0.00%  swapper      [kernel.kallsyms]  [k] rest_init
#      |
#      |--rest_init
#      |
#      |--87.88%--cpu_startup_entry
#      |
#      |--87.83%--arch_cpu_idle
#      |
#      |--87.82%--default_idle
#      |
#      |--87.77%--native_safe_halt
```

输出显示 87% 的时间花在 `rest_init` 函数上。

关于 `perf` 工具的更多使用方式可以参考 IBM 的文章：<https://www.ibm.com/developerworks/cn/linux/l-cn-perf1/index.html>。Java 也有自己专用的 `perf` 工具，不在本文讨论范围。

22.6.4 网络连接和流量

sar `sar` 是用来查看网卡流量的最常用方式，它以字节为单位输出网络传入和传出流量。

```
sar -n DEV 1
```

Average:	IFACE	rxpck/s	txpck/s	rxkB/s	txkB/s	rxcmp/s	txcmp/s	rxmcs/s
Average:	eth0	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Average:	lo	1.52	1.52	0.12	0.50	0.00	0.00	0.00

netstat

`netstat -anp` 可以列出连接的详细信息，并且可以将连接、监听的端口对应到进程。其中 `Recv-Q` 和 `Send-Q` 代表该连接在内核中等待发送和接收的数据长度，单位为字节。例如，发送数据时，`send` 调用将数据从用户态复制到内核态后返回，TCP 协议栈负责将数据发送出去，`Send-Q` 代表了尚未发送出去（未被对端 ACK）的数据量。在未发送完之前，这些数据停留在内核缓冲，原因可能是网络延时，或者对端的滑动窗口限制（例如，对端没有 `read`）。`Recv-Q` 则代表协议栈已完成接收，但尚未被应用层的 `read` 调用从内核态复制到用户态的数据长度。

Active Internet connections (servers and established)

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
tcp	0	0	192.168.122.1:53	0.0.0.0:*	LISTEN	2442/dnsmasq
tcp	0	0	0.0.0.0:22	0.0.0.0:*	LISTEN	1645/sshd





```
tcp          0      0 127.0.0.1:631    0.0.0.0:*    LISTEN      1646/cupsd
```

`netstat -s` 提供了各个协议下的统计信息，例如，活跃连接数、重传次数、`reset` 信息等都非常有用。部分输出信息如下所示。

Tcp:

```
28 active connections openings
1 passive connection openings
24 failed connection attempts
0 connection resets received
1 connections established
9006 segments received
8722 segments send out
6 segments retransmitted
0 bad segments received.
24 resets sent
```

`netstat` 是观察网络连接的常用工具，但是在无法处理海量网络连接的情况下可以用 `ss` 代替。

ss

`ss` 与 `netstat` 功能类似，但适合处理海量连接。

```
sudo ss -anp
```

其返回信息与 `netstat` 类似：

Netid	State	Recv-Q	Send-Q	Local Address:Port	Peer Address:Port
nl	UNCONN	768	0	0:1648	*
nl	UNCONN	0	0	0:-905968623	*
nl	UNCONN	0	0	0:803	*

ifconfig

除了用来查看 IP 地址，还需要留意其中的 `RX/TX errors`、`dropped`、`overruns` 信息，大部分情况下它们没什么问题，但是当网卡流量跑满的时候可能会出现意外。

```
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.16.62.6 netmask 255.255.255.0 broadcast 172.16.62.255
    inet6 fe80::20c:29ff:fe73:3860 prefixlen 64 scopeid 0x20<link>
    ether 01:0c:29:73:38:60 txqueuelen 1000 (Ethernet)
```





```
RX packets 9295 bytes 983197 (960.1 KiB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 9046 bytes 3101391 (2.9 MiB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

sysdig

sysdig 可以分析系统级和进程级许多方面的状况，例如，系统调用、网络统计、文件 I/O 等，现在我们用它捕获某个进程到某个 IP 的网络流量。

通过 `proc.pid` 指定要捕获的进程，`fd.cip` 过滤特定 IP 地址，`evt.buffer contains` 指定要过滤的文本内容，此处我们以捕获 ES 的 ping 请求为例。

```
sysdig -s 4096 -A -c echo_fds fd.type=socket proc.pid=18914 and fd.cip=
10.10.13.13 and evt.buffer contains "internal:discovery/zen/fd/ping"

----- Read 359B from 10.10.13.13:59948->10.10.13.15:9300 (java)

ESaMininternal:discovery/zen/fd/pingt5.0un-9UZ4PS8-K6hF59x1MWAGqKvr6c9RryG
LN-wWzAqvwt5.xx.xx.net
10.10.13.15
t5.xx.xx.net$Tmyclustert3.0LPZFbE2yR6eGFs2rgq9puwP1Hxz0KQR7e-56DnmVFQSA
t3.xx.xx.net
10.10.13.13

t3.xx.xx.net$T
```

sysdig 还可以将捕获的数据录制为文本或二进制格式，关于该工具的更多详细信息可以参考官网：<http://www.sysdig.org/>。

22.7 小结

大部分问题是由于比较简单的因素导致的。系统化地分析问题需要故障能够重现，至少系统正处于异常状态。有些故障只经历了短暂的时间，或者在故障之后无法确认之前都做了哪些操作，或者故障永远都无法在测试环境中重现。这类问题我们可以仔细分析相关流程，在关键的地方添加日志，或者开发特定的接口来了解到底发生了什么。





附录 A 重大版本变化

在考虑是否升级到新版本时，第一个问题经常是：版本变化大吗？这是一个很难回答的问题。背后隐含的问题首先是兼容性方面的：原有的业务是否受到影响？哪些特性被废弃了？使用的查询语法是否变了？我们都有哪些相关模块需要做出调整？调整幅度有多大？这些问题都和业务上的实际使用情况有关，没有固定的答案。除了查看版本变更手册，我们需要根据自己的业务使用情况评估版本变化对自己的影响。

除了版本兼容性方面的问题，另一方面是考虑新版本带来了什么好处？例如，写入速度和查询速度是否变快了？添加了哪些新的特性？这些特性对我来说是否有用？

对于是否需要升级到新版本方面，我们的建议是：不要错过每个大版本。如果因为时间原因或懒得对新版本做出适配和调整，则在后期会越来越难升级，以至于最后基本无法升级。当然，也不必因为每个小版本的更新去升级集群，除非修复了一些影响比较大的 bug。在比较注重稳定性的生产环境中，升级版本可以略微保守：可以选择一个大版本并在社区已经大范围应用检验之后升级。

下面列出一些常用内容涉及的变化，完整的版本变更说明可以参考手册：<https://www.elastic.co/guide/en/elasticsearch/reference/current/breaking-changes.html>。

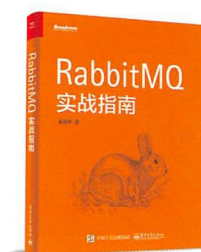
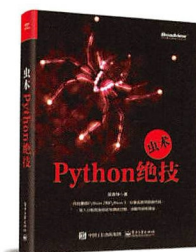
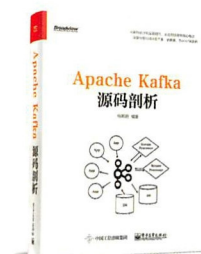
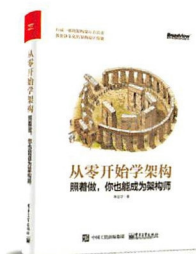
6.x 重大变化

- 每个索引支持一个 `_type`；
- 默认禁用 `_all` 字段；
- 优化了 doc values，占用磁盘空间更少，读写速度更快；
- 模板规则定义方式由 `template` 改为 `index_patterns`；
- 增加了序列 ID，加快索引恢复速度；
- 查询语法变化比较多，请参阅手册。



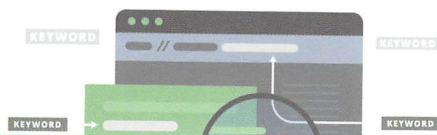


好书力荐



拒绝堆砌臃肿,支持纯正原创

欢迎投稿: chenxm@phei.com.cn



Elasticsearch 源码解析与优化实战

非常高兴看到国内能有一本研究Elasticsearch源码的书出现，Elasticsearch现在已经变得非常流行，掌握这门技术是大势所趋。Elasticsearch虽然上手简单，但是想要成为Elasticsearch的专家可不容易，而阅读本书就是一个很好的开始。

Elastic中文社区创始人 Elastic技术布道师 曾勇 (Medcl)

Elasticsearch作为开源的分布式搜索引擎，近几年开始在国内加速流行。由于常年混迹Elastic中文社区，从社区用户提交的形形色色的问题中，我看到国内用户大多对该技术还缺乏深层次的理解，遇到问题时往往不知所措。比如如何架构集群？如何正确设置集群参数和数据模型？如何优化数据读写？如何应对灾难并迅速恢复数据？

《Elasticsearch源码解析与优化实战》让我眼前一亮！本书内容整理自作者的ES源码分析系列博客，而我也曾通过阅读该系列博客，加深了对ES内部运作机理的认识。如果你希望从ES工程师转变为ES技术专家、架构师，相信这本书会给你带来极大的帮助。

携程旅行网技术保障部 系统研发总监 吴晓刚

ES作为一个大小公司都广泛使用的开源搜索引擎，其中文方面的系统的资料却少得可怜。当初我们公司遇到ES的问题时，也找不到太好的资料，只好求助朋友。

有幸第一时间拜读本书样稿，书中系统和详细地介绍了ES的概念、各个流程、优化方案、应用实践、各种故障诊断等。难能可贵的是，每一部分都有作者的思考与总结。发现当初在ES上踩过的各种坑，本书中都有详细的解释。

阿里巴巴前研究员 阿里妈妈前技术部负责人 吴雪军



博文视点Broadview



新浪微博
weibo.com

@博文视点Broadview



责任编辑：陈晓猛

封面设计：李玲

上架建议：计算机/分布式搜索

ISBN 978-7-121-35216-4



9 787121 352164 >

定价：89.00元

版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF